Practical Python Artificial Intelligence Programming

by Mark Watson

VISIT…

LANZAROTE Caliente.com

Practical Python Artificial Intelligence Programming

Mark Watson

This book is for sale at http://leanpub.com/pythonai

This version was published on 2023-02-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Cover Material, Copyright, and License

This eBook will be updated occasionally so please periodically check the leanpub.com web page for this book* for updates.

Please visit my website† and follow me on social media.

*https://leanpub.com/pythonai
†http://markwatson.com

Preface

This book is intended, dear reader, to show you a wide variety of practical AI techniques and examples, and to be a jumping off point when you discover things that interest you or may be useful in your work. A common theme here is covering AI programming tasks that used to be difficult or impossible but are now much simpler using deep learning, of at least possible. I also cover a wide variety on non-deep learning material including a chapter on Symbolic AI that has historic interest and some current practical value.

I try to update my books at least once a year so when purchasing on Leanpub please indicatate that you want to be notified when new editions are available. Updates to new editions are free for my Leanpub books.

My career developing AI applications and tools began in 1982. Until the advent of breakthroughs in deep learning around 2010 most of my development work was in Common Lisp, Java, and C++. My language preference changed when I started spending most of my time creating deep learning models. Python has the most tooling, libraires, and frameworks for deep learning so as a practical matter I have adopted Python as a primary programming language. That said I still also heavily use Common Lisp, Haskell, Swift, and Scheme. I recommend not having an “always use one programming language” mindset.

Why this book? Some of what I cover here has already been covered in the Common Lisp, Java, Clojure and Haskell artificial intelligence books I have previously written. My goal here is to prioritize more material on deep learning while still lightly covering classical machine learning, knowledge representation, information gathering, and semantic web/linked data theory and applications. We also cover knowledge representation, including: classic systems like Soar and Prolog, constraint programming, and the practical use of relational and graph data stores. Much of my work involves natural language processing (NLP) and associated techniques for processed unstructured data and we will cover this material in some depth.

Why Python? Python is a very high level language that is easily readable by other programmers. Since Python is one of the most popular programming languages there are many available libraries and frameworks. The best code is code that we don’t have to write ourselves as long as third party code is open source so we can read and modify it if needed. Another reason to use Python, that we lean heavily on in this book, is using pre-trained deep learning models that are wrapped into Python packages and libraries.

About the Author

I have written over 20 books, I have over 50 US patents, and I have worked at interesting companies like Google, Capital One, SAIC, Mind AI, and others. You can read all of my recent books (including

this book) for free on my web site https://markwatson.com. If I had to summarize my career the short take would be that I have had a lot of fun and enjoyed my work. I hope that what you learn here will be both enjoyable and help you in your work.

If you would like to support my work please consider purchasing my books on Leanpub* and star my git repositories that you find useful on GitHub†. You can also interact with me on social media on Mastodon‡ and Twitter§.

Using the Example Code

The example code that I have written for this book is Apache 2 licensed so feel free to reuse it. I also use several existing open source packages and libraries in the examples that use liberal-use licenses (I link GitHub repositories, so check the licenses for applicability in your projects). Most of the deep learning examples and the few "classic" machine learning examples in this book are available as Jupyter notebooks in the **jupyter_notebooks** directory that can be run as-is on Google Colab¶ (or install Jupyter locally on your laptop) or the equivalent Python source files are in the **deep-learning** directory. One advantage of using Colab is that most of the required libraries are pre-installed.

The examples for this book are in the GitHub repository https://github.com/mark-watson/PythonPracticalAIBookCode.

A few of the examples use APIs from Hugging Face and OpenAI's GPT-3. I assume that you have signed up and have access keys that should be available in the environment variables **HF_API_TOKEN** and **OPENAI_KEY**. If you don't want to sign up for these services I still hope that you enjoy reading the sample code and example output.

The GitHub repository https://github.com/mark-watson/PythonPracticalAIBookCode for my code examples will occasionally contain subdirectories containing code not in the current edition of this book but are likely to appear in future editions. These subdirectories contain a file named **NOT_YET_IN_BOOK.md**. I plan on releasing new editions of this book in the future.

I have not written original example code for all of the material in this book. In some cases there are existing libraries for such tasks as recommendation systems and generating images from text where I reference third party examples and discuss how and why you might want to use them.

Book Cover

I live in Sedona Arizona. I have been fortunate to have visited close to one hundred ancient Native American Indian sites in the Verde Valley. I took the cover picture at one of these sites.

*https://leanpub.com/u/markwatson
†https://github.com/mark-watson?tab=repositories&q=&type=public
‡https://mastodon.social/@mark_watson
§https://twitter.com/mark_l_watson
¶https://colab.research.google.com

This picture shows me and my wife Carol who helps me with book production.

Acknowledgements

I would like to thank my wife Carol Watson who edits all of my books.

I would like to thank the following readers who reported errors or typos in this book: Ryan O'Connor (typo).

Part I - Getting Started

In the next three chapters we setup our Python programming environment, use the Scikit-learn library for one classic machine learning example, and experiment with what is often called Good Old Fashioned Symbolic AI (GOFAI).

Python Development Environment

I don't use a single development setup for Python. Here I will describe the tools I use and in what contexts I use them.

When you are setting up a Python development environment there are a few things to consider in order to ensure that your environment is set up correctly and is able to run your code correctly. Here are a few pieces of advice to help you get started (web references: getting started*, testing†, virtual environments‡, and packaging§):

- Use Git or other version control systems to manage your codebase and keep track of changes. This will make it easier to collaborate with other developers and keep your code organized. I will not cover Git here so read a good tutorial¶ if you have not used it before.
- Use a virtual environment to isolate your development environment and dependencies from the rest of your system. This will make it easier to manage your dependencies and avoid conflicts with other software on your system.
- Use pip or another package manager to manage your dependencies and install packages. This will make it easier to install and update packages, and will also help to make sure that you have the correct versions of packages installed.
- For large programs use an IDE such as PyCharm, VSCode or any other to write, run and debug your code. For short Python programs I usually skip using an IDE and instead use Emacs + Python mode.
- Test your code: Be sure to test your code. Use testing frameworks such as unittest, nose or pytest to automate your testing process.
- Keep your environment and dependencies up-to-date, to ensure that you are using the latest versions of packages and that your code runs correctly.
- Add comments and documentation to your code so that other developers (and you!!) can understand what your code is doing and how it works. Even if you are working on personal projects your "future you" will thank you for adding comments when you need to revisit your own code later.

When developing using Python, or any other programming language, you will find a lot of advice in books and on the web. While I really enjoy tweaking my development environment to get it "just right," I try to minimize the time I invest in this tuning process. This may also work for you: when you are tired near the end of a workday and you might not be at your best for developing new algorithms or coding then use that time to read about and experiment with your development environment, learn new features of your favorite programming languages, or try a new text editor.

*https://www.python.org/about/gettingstarted/
†https://docs.python.org/3/library/unittest.html
‡https://docs.python.org/3/tutorial/venv.html
§https://packaging.python.org/en/latest/tutorials/managing-dependencies/
¶https://git-scm.com/docs/gittutorial

Managing Python Versions and Libraries

There are several tools for managing Python versions and installed libraries. Here I discuss the tools I use.

Anaconda

The Anaconda Corporation maintains open source tools and provides enterprise support. I use their MiniConda package manager to install different environments containing specific versions of Python and specific libraries for individual projects.

Pause and read the online documentation.*

I use Anaconda for managing complex libraries and frameworks for machine learning and deep learning that often have different requirements if a GPU is available. Installing packages takes longer with Anaconda compared to other options but Anaconda's more strict package dependency analysis ends up saving me time when running deep learning on my laptop or my servers.

Here is an example for setting up the environment for the examples in the next chapter:

```
conda create -n ml python=3.10
conda activate ml
conda install scikit-learn
```

Google Colab

Google Colab, short for Colaboratory, is a free cloud-based platform for data science and machine learning developed by Google. It allows users to write and execute code in a web-based environment, with support for Jupyter notebooks and integration with other Google services such as Google Drive.

Google Colab lets me get set up for working on machine learning and deep learning projects without having to install Python and dependencies on my laptops or servers. It allows me to juggle projects with different requirements while keeping each project separate.

In technical terms, Colab is a Jupyter notebook environment that runs on a virtual machine in the cloud, with access to powerful hardware such as GPUs and TPUs. The virtual machine is pre-configured with popular data science libraries and tools such as TensorFlow, PyTorch, and scikit-learn, making it easy to get started with machine learning and deep learning projects.

Colab also includes features such as code execution, debugging, and version control, as well as the ability to share and collaborate on notebooks with others. Additionally, it allows you to mount your google drive as a storage, which can be useful for large data sets or models.

*https://docs.anaconda.com/anaconda/user-guide/index.html

Overall, Google Colab is a useful tool for data scientists and machine learning engineers as it provides a convenient and powerful environment for developing and running code, and for collaboration and sharing of results.

Since 2015 most of my work has been centered around deep learning and Google Colab saves me a lot of time and effort. I pay for Colab Pro ($10/month) to get higher priority for using GPUs and TPUs but this is not strictly necessary (many users just use the free tier).

venv

venv creates a subdirectory for a project's installed Python executable files and libraries. You can name this subdirectory whatever you like (I like the name "venv"):

I sometimes use venv[*] for my Python related work that is not machine learning or deep learning. **venv** is used to create isolated virtual Python development environments for each project, for example:

```
1 $ python3 -m venv venv
2 $ source venv/bin/activate
3 $ pip install minizinc sparqlwrapper
```

We activate the newly created virtual environment in line 2 and install two Python packages into the new virtual environment in line 3.

Editors and IDEs

I assume that you already have a favorite Python editor or IDE. Here I will just mention what I use and why I choose different tools for different types of tasks.

I like VSCode when I am working on a large Python project that has many source files scattered over many directories and subdirectories because I find navigation and code browsing is fast and easy.

I prefer Emacs with **python-mode** for most of my work that consists of smaller projects where all code is in either a single or just a few source files. For larger projects I sometimes use Emacs with *treemacs* for rapid navigation between files. I especially like the interactive coding style with Emacs and **emacs-mode** because it is simple to load an entire file, re-load a changed function definition, etc., and work interactively in the provided REPL.

I sometimes use the PyCharm IDE. PyCharm also has excellent rapid code navigation support and is generally full featured. Until about a year ago I used PyCharm for most of my development work but lately I have gone back to using Emacs and **python-mode** as my main daily driver.

[*] https://docs.python.org/3/library/venv.html

Code Style

I recommend installing and configuring black[*] and then install isort[†]. Some Python developers prefer integrating black and isort with their editors or IDEs to reformat Python code on every file save but I prefer using a **Makefile** target called **tidy** to run black and isort on all python source files in a project. Both tools are easily installed:

```
pip install black
pip install isort
```

You can also run them manually:

```
$ black *.py
reformatted us_states.py

All done!
1 file reformatted, 1 file left unchanged.
$ isort *.py
$
```

[*] https://black.readthedocs.io/
[†] https://pycqa.github.io/isort/

"Classic" Machine Learning

"Classic" Machine Learning (ML) is a broad field that encompasses a variety of algorithms and techniques for learning from data. These techniques are used to make predictions, classify data, and uncover patterns and insights. Some of the most common types of classic ML algorithms include:

- Linear regression: a method for modeling the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data.
- Logistic regression: a method for modeling the relationship between a binary dependent variable and one or more independent variables by fitting a logistic function to the observed data.
- Decision Trees: a method for modeling decision rules based on the values of the input variables, which are organized in a tree-like structure.
- Random Forest: a method that creates multiple decision trees and averages the results to improve the overall performance of the model
- K-Nearest Neighbors (K-NN): a method for classifying data by finding the K-nearest examples in the training data and assigning the most similar common class among them.
- Naive Bayes: a method for classifying data based on Bayes' theorem and the assumption of independence between the input variables.

We will be covering a very small subset of "classic" ML, and then dive deeper into Deep Learning in later chapters. Deep Learning differs from classic ML in several ways:

- Scale: Classic ML algorithms are typically designed to work with small to medium-sized datasets, while deep learning algorithms are designed to work with large-scale datasets, such as millions or billions of examples.
- Architecture: Classic ML algorithms have a relatively shallow architecture, with a small number of layers and parameters, while deep learning algorithms have a deep architecture, with many layers and millions or billions of parameters.
- Non-linearity: Classic ML algorithms are sometimes linear, (i.e., the relationship between the input and output is modeled by a linear equation), while deep learning algorithms are non-linear, (i.e., the relationship is modeled by a non-linear function).
- Feature extraction: "Classic" ML requires feature extraction, which is the process of transforming the raw input data into a set of features that can be used by the algorithm. Deep learning can automatically learn features from raw data, so it does not usually require too much separate effort for feature extraction.

So, Deep Learning is a subfield of machine learning that is focused on the design and implementation of artificial neural networks with many layers which are capable of learning from large-scale and complex data. It is characterized by its deep architecture, non-linearity, and ability to learn features from raw data, which sets it apart from "classic" machine learning algorithms.

Example Material

Here we cover just a single example of what I think of as "classic machine learning" using the scikit-learn* Python library. Later we cover deep learning in three separate chapters. Deep learning models are more general and powerful but it is important to recognize the types of problems that can be solved using the simpler techniques.

They only requirements for this chapter is **pip install scikit-learn pandas**.

Please note that the content in this book is heavily influenced by what I use in my own work. I mostly use deep learning so its coverage comprises half this book. For this classic machine learning chapter I only use a classification model. I will not be covering regression or clustering models.

We will use the same Wisconsin cancer dataset for both the following classification example and a deep learning classification example in a later chapter. Here are the first few rows of the file **labeled_cancer_data.csv**:

Cl.thickness	Cell.size	Cell.shape	Marg.adhesion	Epith.c.size	Bare.nuclei	Bl.cromatin	Normal.nucleoli	Mitoses	Class
5	1	1	1	2	1	3	1	1	0
3	1	1	1	2	2	3	1	1	0
4	1	1	3	2	1	3	1	1	0
1	1	1	1	2	10	3	1	1	0
2	1	1	1	2	1	1	1	5	0
1	1	1	1	1	1	3	1	1	0
5	3	3	3	2	3	4	4	1	1
8	7	5	10	7	9	5	5	4	1

The last column **class** indicates the class of the sample, 0 for non-malignant and 1 for malignant. The scikit-learn library has high level and simple to use utilities for reading CSV (spreadsheet) data and for preparing the data for training and testing. I don't use these utilities here because I am reusing the data loading code from the later deep learning example.

We will use the Pandas library and if you have not used Pandas before you might want to reference the Pandas documentation†. Here is a summary of why Pandas is generally useful: it is a popular Python library for data manipulation and analysis that provides data structures and data analysis tools for working with structured data (often spreadsheet data). One of the key data structures in Pandas is the DataFrame, which is a two-dimensional table of data with rows and columns.

A DataFrame is essentially a labeled, two-dimensional array, where each column has a name and each row has an index. DataFrames are similar to tables in a relational database or data in a spreadsheet. They can be created from a variety of data sources such as CSV, Excel, SQL databases,

*https://scikit-learn.org/stable/
†https://pandas.pydata.org/docs/

or even Python lists and dictionaries. They can also be transformed, cleaned, and manipulated using a wide range of built-in methods and functionalities.

Pandas DataFrames provide a lot of functionalities to handle and process data. The most common ones are:

- Indexing: data can be selected by its label or index.
- Filtering: data can be filtered by any condition.
- Grouping: data can be grouped by any column.
- Merging and joining: data can be joined or merged with other data.
- Data type conversion: data can be converted to any data type.
- Handling missing data: data can be filled in or removed based on any condition.

DataFrames are widely used in data science and machine learning projects for loading, cleaning, processing, and analyzing data. They are also used for data visualization, data preprocessing, and feature engineering tasks.

Listing of **load_data.py**:

```
1  import pandas
2
3  def load_data():
4
5      train_df = pandas.read_csv("labeled_cancer_data.csv")
6      test_df = pandas.read_csv("labeled_test_data.csv")
7
8      train = train_df.values
9      X_train = train[:,0:9].astype(float) # 9 inputs
10     print("Number training examples:", len(X_train))
11     # Training data: one target output (0 for no cancer, 1 for malignant)
12     Y_train = train[:,-1].astype(float)
13     test = test_df.values
14     X_test = test[:,0:9].astype(float)
15     Y_test = test[:,-1].astype(float)
16     print("Number testing examples:", len(X_test))
17     return (X_train, Y_train, X_test, Y_test)
```

In line 8 we fetch the values array from the data frame and in line 9 we copy all rows of data, skipping the last column (target classification we want to be able to predict) and converting all data to floating point numbers. In line 14 we copy just the last column of the training data array for use as the target classification.

Classification Models using Scikit-learn

Classification is a type of supervised machine learning problem where the goal is to predict the class or category of an input sample based on a set of features. The goal of a classification model is to learn a mapping from the input features to the output class labels.

Scikit-learn (sklearn) is a popular Python library for machine learning that is effective for several reasons (partially derived from the Skikit-learn documentation*):

- Scikit-learn provides a consistent and intuitive API for a wide range of machine learning models, which makes it easy to switch between different algorithms and experiment with different approaches.
- Scikit-learn includes a large collection of models for supervised, unsupervised, and semi-supervised learning (e.g., linear and non-linear models, clustering, and dimensionality reduction).
- Scikit-learn includes a wide range of preprocessing and data transformation tools, such as feature scaling, one-hot encoding, feature extraction, and more, that can be used to prepare and transform the data before training the model.
- Scikit-learn provides a variety of evaluation metrics, such as accuracy, precision, recall, F1-score, and more, that can be used to evaluate the performance of a model on a given dataset.
- Scikit-learn is designed to be scalable, which means it can handle large datasets and high-dimensional feature spaces. It also provides tools for parallel and distributed computing, which can be used to speed up the training process on large datasets.
- Scikit-learn is designed to be easy to use, with clear and concise code, a well-documented API, and plenty of examples and tutorials that help users get started quickly.
- Scikit-learn is actively developed and maintained, with regular updates and bug fixes, and a large and active community of users and developers who contribute to the library and provide support through mailing lists, forums, and other channels.

All these features make Scikit-learn a powerful and widely used machine learning library for various types of machine learning tasks.

```python
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix

from load_data import load_data

(X_train, Y_train, X_test, Y_test) = load_data()

# Remove mean and scale to unit variance:
```

*https://scikit-learn.org/0.21/documentation.html

```
scaler1 = StandardScaler()
scaler2 = StandardScaler()

X_train = scaler1.fit_transform(X_train)
X_test = scaler2.fit_transform(X_test)

# Use the KNN classifier to fit data:
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(X_train, Y_train)

# Predict y data with classifier:
y_predict = classifier.predict(X_test)

# Print results:
print(confusion_matrix(Y_test, y_predict))
print(classification_report(Y_test, y_predict))
```

We can now train and test the model and evaluate how accurate the model is. In reading the following output, you should understand a few definitions. In machine learning, precision, recall, F1-score, and support are all metrics used to evaluate the performance of a classification model, specifically in regards to binary classification:

- Precision: the proportion of true positive predictions out of the total of all positive predictions made by the model. It is a measure of how many of the positive predictions were actually correct.
- Recall: the proportion of true positive predictions out of all actual positive observations in the data. It is a measure of how well the model is able to find all the positive observations.
- F1-score: the harmonic mean of precision and recall. It is a measure of the balance between precision and recall and is generally used when you want to seek a balance between precision and recall.
- Support: number of observations in each class.

These metrics provide an overall view of a model’s performance in terms of both correctly identifying positive observations and avoiding false positive predictions.

```
$ python classification.py
Number training examples: 677
Number testing examples: 15
[[7 0]
 [1 7]]
              precision    recall  f1-score   support

         0.0       0.88      1.00      0.93         7
         1.0       1.00      0.88      0.93         8

    accuracy                           0.93        15
   macro avg       0.94      0.94      0.93        15
weighted avg       0.94      0.93      0.93        15
```

Classic Machine Learning Wrap-up

I have already admitted my personal biases in favor of deep learning over simpler machine learning and I proved that by using perhaps only 1% of the functionality of Scikit-learn in this chapter.

Symbolic AI

When I started my paid career as an AI practitioner in 1982 my company bought me a Xerox 1108 Lisp Machine and I spent every spare moment I had working through two books by Patrick Winston that I had purchased a few years earlier: "Lisp" and "Artificial Intelligence." This material was mostly what is now called symbolic AI or good old fashioned AI (GOFAI). The material in this chapter is optional for modern AI developers but I recently wrote the Python examples listed below when I was thinking of how different knowledge representation is today compared to 40 years ago. Except for the material using Python + Swi-Prolog, and Python + the MiniZinc constraint satisfaction system there is nothing in this chapter that I would consider using today for work but you might enjoy the examples anyway. After this chapter we will bear down on deep learning, information organization using RDF and property graph data stores.

I do not implement three examples in this chapter in "pure Python," rather, I use the Python bindings for three well known tools that are implemented in C/C++:

- Swi-Prolog is a Prolog system that has many available libraries for a wide variety of tasks.
- Soar Cognitive Architecture is a flexible and general purpose reasoning and knowledge management system for building intelligent software agents.
- MiniZinc is a powerful Constraint Satisfaction System.

The material in this chapter is optional for the modern AI practitioner but I hope you find it interesting.

Comparison of Symbolic AI and Deep Learning

Symbolic AI, also known as "good old-fashioned AI" (GOFAI), is a form of artificial intelligence that uses symbolic representations and logical reasoning to solve problems. It is based on the idea that a computer can be programmed to manipulate symbols in the same way that humans manipulate symbols in their minds: an example of this is the process of logical reasoning.

Symbolic AI systems can consist of sets of rules, facts, and procedures that are used to represent knowledge and a reasoning engine that uses these symbolic representations to make inferences and decisions. Some examples of symbolic AI systems include expert systems, other types of rule-based systems, and decision trees. These systems are typically based on a set of predefined rules and the performance of the system is based on the knowledge manually encoded (or it can be learned) in these rules. Symbolic AI is largely non-numerical.

In comparison deep learning, which uses numerical representations (high dimensional matrices, or tensors, containing floating point data), is a subset of machine learning that uses neural networks

with many multiple layers (the term "deep" comes from the idea of having dozens or even hundreds of layers, which differs from early neural network models comprised of just a few layers) to learn from data and make predictions or decisions. The basic building blocks of both simple neural models and deep learning models are artificial neurona, which are a simple mathematical function that receives input, applies a transformation, and produces an output. Neural networks can learn to perform a wide variety of tasks, such as image recognition, natural language processing, and game playing, by adjusting the weights of the neurons.

The key difference is that, while Symbolic AI relies on hand-coded rules and logical reasoning, deep learning relies on learning from data. Symbolic AI systems typically have a high level of interpretability and transparency, as the rules and knowledge are explicitly encoded and can be inspected by humans, while deep learning models are often considered "black boxes" due to their complexity and the difficulty of understanding how they arrived at a decision.

So, Symbolic AI uses symbolic representations and logical reasoning while deep learning uses neural networks to learn from data. The first one is more interpretable but less flexible, while the second one is more flexible, much more powerful for most applications, but is less interpretable.

We will start with one "pure Python" example in the next section.

Implementing Frame Data Structures in Python

Most of my learning experiments and AI projects in the early 1980s were built from scratch in Common Lisp and nested frame data structures were a common building block. Here we allow three types of data to be stored in frames:

- Numbers
- Strings
- Other frames

We write a general Python class **Frame** that supports creating frames and converting a frame, including deeply nested frames, into a string representation. We also write a simple Python class **BookShelf** as a container for frames that supports searching for any frames containing a string value.

```python
# Implement Lisp-like frames in Python

class Frame():
    frame_counter = 0
    def __init__(self, name = ''):
        Frame.frame_counter += 1
        self.objects = []
        self.depth = 0
        if (len(name)) == 0:
            self.name = f"Frame:{Frame.frame_counter}"
        else:
            self.name = f'"{name}"'

    def add_subframe(self, a_frame):
        a_frame.depth = self.depth + 1
        self.objects.append(a_frame)

    def add_number(self, a_number):
        self.objects.append(a_number)

    def add_string(self, a_string):
        self.objects.append(a_string)

    def __str__(self):
        indent = "  " * self.depth * 2
        ret = indent + f"<Frame {self.name}>\n"
        for frm in self.objects:
            if isinstance(frm, (int, float)):
                ret = ret + indent + '    ' + f"<Number {frm}>\n"
            if isinstance(frm, str):
                ret = ret + indent + '    ' + f'<String "{frm}">\n'
            if isinstance(frm, Frame):
                ret = ret + frm.__str__()
        return ret

f1 = Frame()
f2 = Frame("a sub-frame")
f1.add_subframe(f2)
f1.add_number(3.14)
f2.add_string("a string")
print(f1)
f2.add_subframe(Frame('a sub-sub-frame'))
print(f1)
```

```

class BookShelf():

    def __init__(self, name = ''):
        self.frames = []

    def add_frame(self, a_frame):
        self.frames.append(a_frame)

    def search_text(self, search_string):
        ret = []
        for frm in self.frames:
            if frm.__str__().index(search_string):
                ret.append(frm)
        return ret

bookshelf = BookShelf()
bookshelf.add_frame(f1)
search_results = bookshelf.search_text('sub')
print("Search results: all frames containing 'sub':")
for rs in search_results:
    print(rs)
```

The implementation of the class **Frame** is straightforward. The init method **init** defines a list of contained objects (or frames), sets the default frame nesting depth to zero, and assigns a readable name. There are three separate methods to add subframes, numbers, and strings.

The method **str** ensures that when we print a frame that the output is human readable and visualizes frame nesting.

Here is some output:

```
$ python
>>> from frame import Frame Bookshelf
>>> f1 = Frame()
>>> f2 = Frame("a sub-frame")
>>> f1.add_subframe(f2)
>>> f1.add_number(3.14)
>>> f2.add_subframe(Frame('a sub-sub-frame'))
>>> print(f1)
<Frame Frame:4>
  <Frame "a sub-frame">
    <Frame "a sub-sub-frame">
  <Number 3.14>
```

```
>>> bookshelf = BookShelf()
>>> bookshelf.add_frame(f1)
>>> search_results = bookshelf.search_text('sub')
>>> for rs in search_results:
...   print(rs)
...
<Frame Frame:4>
  <Frame "a sub-frame">
    <Frame "a sub-sub-frame">
  <Number 3.14>
```

I my early AI experiments in the 1980s I would start with implementing a simple frame library and extend it for the two types of applications that I worked on: Natural Language Processing (NLP) and planning systems.

I no longer use frames, preferring the use of off the shelf graph databases that we will cover in a later chapter. Graphs can represent a wider range of data representations because frames represent tree structured data and graphs are more general purpose than trees.

Use Predicate Logic by Calling Swi-Prolog

Please skip this section if you either don't know how to program in Prolog or if you have no interest in learning Prolog. I have a writing project for a book titled Prolog for AI applications that is a work in progress. When that book is released I will add a link here. Before my Prolog book is released, please use Sheila McIlraith's Swi-Prolog tutorial[*]. It was written for her students and is a good starting point. You can also use the official Swi-Prolog manual† for specific information. I will make this section self-contained if you just want to read the material without writing your own Python + Prolog applications.

You can start by reading the documentation for setting up Swi-Prolog so it can be called from Python‡.

I own many books on Prolog but my favorite that I recommend is "The Art Of Prolog"§ by Leon S. Sterling and Ehud Y. Shapiro. Here are a few benefits to using Prolog:

- Prolog is a natural fit for symbolic reasoning: Prolog is a logic programming language, which means that it is particularly well-suited for tasks that involve symbolic reasoning and manipulation of symbols. This makes it an excellent choice for tasks such as natural language processing, knowledge representation, and expert systems.

[*] https://www.cs.toronto.edu/~sheila/324/f05/tuts/swi.pdf
† https://www.swi-prolog.org/pldoc/doc_for?object=manual
‡ https://www.swi-prolog.org/pldoc/man?section=mqi-python-installation
§ https://mitpress.mit.edu/9780262691635/the-art-of-prolog/

- Prolog has built-in support for logic and rule-based systems which makes it easy to represent knowledge and perform inferences. The syntax of Prolog is based on first-order logic, which means that it is similar to the way humans think and reason.
- Prolog provides a high-level and expressive syntax, which makes it relatively easy to write and understand code. The declarative nature of Prolog also makes it more readable and maintainable than some other languages.
- Prolog provides automatic backtracking and search, which makes it easy to implement algorithms that require searching through large spaces of possible solutions.
- Prolog has a set of built-in predicates and libraries that support natural language processing, which made it an ideal choice for tasks such as natural language understanding and generation. Now deep learning techniques are the more effective technology or NLP.
- Prolog has a large community of users and developers which means that there are many open source libraries and tools available for use in Prolog projects.
- Prolog can be easily integrated with other programming languages, such as Python, C, C++, Common Lisp, and Java. This integration with other languages makes Prolog a good choice for projects that require a combination of different languages.
- Prolog code is concise, which means less lines of code to write. Also, some programmers find that the declarative nature of the language makes it less prone to errors.

Using Swi-Prolog for the Semantic Web, Fetching Web Pages, and Handling JSON

In several ways, reading the historic Scientific American Article from 2001 The Semantic Web - A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities* by Tim Berners-Lee, James Hendler, and Ora Lassila changed my life. I spent a lot of time experimenting with the Swi-Prolog **semweb** library that I found to be the easiest way to experiment with RDF. We will cover the open source Apache Jena/Fuseki RDF data store and query engine in a later chapter. The Common Lisp and Semantic Web tools company Franz Inc.† very kindly subsidized my work writing two Semantic Web books that cover Common Lisp, Java, Clojure, and Scala (available as downloadable PDFs on my web site‡). Writing these books lead directly to being invited to work at Google for a project using their Knowledge Graph.

Here I mention how to load the **semweb** library and refer you to the library documentation§:

*https://www-sop.inria.fr/acacia/cours/essi2006/Scientific%20American_%20Feature%20Article_%20The%20Semantic%20Web_%20May%202001.pdf

†https://franz.com

‡https://markwatson.com

§https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/semweb.html%27)

```
?- use_module(library(semweb/rdf_db)).
?- use_module(library(semweb/sparql_client)).
?- sparql_query('select * where { ?s ?p "Amsterdam" }',
                Row,
                host('dbpedia.org'), path('/sparql/')]).
```

The output is:

```
Row = row('http://dbpedia.org/resource/Anabaptist_riot',
          'http://dbpedia.org/ontology/combatant') ;)
Row = row('http://dbpedia.org/resource/Thirteen_Years_War_(1454-1466)',
          'http://dbpedia.org/ontology/combatant') ;)
Row = row('http://dbpedia.org/resource/Womens_Euro_Hockey_League',
          'http://dbpedia.org/ontology/participant') ;)
```

Prolog is a flexible and general purpose language that is used to write compilers, handle text processing, etc. One common thing that I need no matter what programming language I use is fetching content from the web. Here is a simple example you can try to perform a HTTP GET operation and echo the fetched data to standard output:

```
use_module(library(http/http_open)).
http_open('https://markwatson.com', In, []),
   copy_stream_data(In, user_output),
   close(In).
```

Similarly, handling JSON data is a common task so here is an example for doing that:

```
?- use_module(library(http/json)). % to enable json_read_dict/2
?- FPath = 'test.json', open(FPath, read, Stream), json_read_dict(Stream, Dicty).
```

Python and Swi-Prolog Interop

You need to install the Python bridge library that is supported by the Swi-Prolog developers:

```
 pip install swiplserver
```

I copied a Prolog program from the Swi-Prolog documentation* to calculate how to eight queens on a chess board in such a way that no queen can capture another queen. Here I get three results by entering the semicolon key to get another answer or the period key to stop:

*https://www.swi-prolog.org/pldoc/man?section=clpfd-n-queens

```
$ swipl

?- use_module(library(clpfd)). /* constraint satisfaction library */
true.

?- [n_queens]. /* load the file n_queens.pl */
true.

?- n_queens(8, Qs), label(Qs).
Qs = [1, 5, 8, 6, 3, 7, 2, 4] ;
Qs = [1, 6, 8, 3, 7, 4, 2, 5] ;
Qs = [1, 7, 4, 6, 8, 2, 5, 3] .
```

The source code to **n_queens.pl** is included in the examples directory **swi-prolog** for this book. This example was copied from the Swi-Prolog documentation. Here is Python code to use this Prolog example:

```
from swiplserver import PrologMQI
from pprint import pprint

with PrologMQI() as mqi:
    with mqi.create_thread() as prolog_thread:
        prolog_thread.query("use_module(library(clpfd)).")
        prolog_thread.query("[n_queens].")
        result = prolog_thread.query("n_queens(8, Qs), label(Qs).")
        pprint(result)
        print(len(result))
```

We can run this example to see all 92 possible answers:

```
 $ p n_queens.py
[{'Qs': [1, 5, 8, 6, 3, 7, 2, 4]},
 {'Qs': [1, 6, 8, 3, 7, 4, 2, 5]},
 {'Qs': [1, 7, 4, 6, 8, 2, 5, 3]},
 {'Qs': [1, 7, 5, 8, 2, 4, 6, 3]},
 {'Qs': [2, 4, 6, 8, 3, 1, 7, 5]},
 ...
 ]
 92
```

Here I call the Swi-Prolog system synchronously; that is, each call to **prolog_thread.query** waits until the answers are ready. If you can also run long running queries asynchronously then please read the instructions online*.

*https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/mqi.html%27)

In the last example we simply ran an existing Prolog program from Python. Now let's look at an example for asserting facts and Prolog rules from a Python script. First we look at a simple example of Prolog rules, asserting facts, and applying rules to facts. We will use the Prolog source file **family.pl**:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :-
  parent(X, Y),
  parent(Y, Z).
```

Before using a Python script, let's run an example in the Swi-Prolog REPL (in line 2, running **[family]**. loads the Prolog source file **family.pl**):

```
1 $ swipl
2 ?- [family].
3 true.
4
5 ?- assertz(mother(irene, ken)).
6 true.
7
8 ?- assertz(father(ken, ron)).
9 true.
10
11 ?- grandparent(A,B).
12 A = irene,
13 B = ron ;
14 false.
```

When running Prolog queries you may get zero, one, or many results. The results print one at a time. Typing the semicolon character after a result is printed requests that the next result be printed. Typing a period after a result is printed lets the Prolog system know that you don't want to see any more of the available results. When I entered a semicolon character in line 13 after the first result is printed, the Prolog system prints *false* because no further results are available.

Now we can write a Python script **family.py** that loads the Prolog rules file **family.pl**, asserts facts, run Prolog queries, and get the results back to the Python script:

```python
from swiplserver import PrologMQI
from pprint import pprint

with PrologMQI() as mqi:
    with mqi.create_thread() as prolog_thread:
        prolog_thread.query("[family].")
        print("Assert a few initial facts:")
        prolog_thread.query("assertz(mother(irene, ken)).")
        prolog_thread.query("assertz(father(ken, ron)).")
        result = prolog_thread.query("grandparent(A, B).")
        pprint(result)
        print(len(result))
        print("Assert another test fact:")
        prolog_thread.query("assertz(father(ken, sam)).")
        result = prolog_thread.query("grandparent(A, B).")
        pprint(result)
        print(len(result))
```

The output looks like:

```
$ python family.py
Assert a few initial facts:
[{'A': 'irene', 'B': 'ron'}]
1
Assert another test fact:
[{'A': 'irene', 'B': 'ron'}, {'A': 'irene', 'B': 'sam'}]
2
```

Swi-Prolog is still under active development (the project was started in 1985) and used for new projects. If the declarative nature of Prolog programming appeals to you then I urge you to take the time to integrate Swi-Prolog into one of your Python-based projects.

Swi-Prolog and Python Deep Learning Interop

Good use cases for Python and Prolog applications involve using Python code to fetch and process data that is imported to Prolog. Applications can then use Prolog's reasoning and other capabilities.

Here we look at a simple example that:

- Uses the Firebase Hacker News APIs to fetch most recent stories.
- Uses the spaCy library deep learning based NLP model to identify organization and peoples names from articles.

- Assert as Prolog facts terms like **organization(Name, URI)***.

We will cover the spaCy library in depth later. For the purposes of this example, please consider spaCy as a "black box."

The following listing shows the Python script **hackernews.py**:

```
from urllib.request import Request, urlopen
import json
from bs4 import BeautifulSoup
from pprint import pprint

from swiplserver import PrologMQI

import spacy
try:
    spacy_model = spacy.load("en_core_web_sm")
except:
  from os import system
  system("python -m spacy download en_core_web_sm")
  spacy_model = spacy.load('en_core_web_sm')

LEN = 100 # larger amount of text is more expensive for OpenAI APIs

def get_new_stories(anAgent={'User-Agent': 'PythonAiBook/1.0'}):
    req = Request("https://hacker-news.firebaseio.com/v0/newstories.json",
                  headers=anAgent)
    httpResponse = urlopen(req)
    data = httpResponse.read()
    #print(data)
    ids = json.loads(data)
    #print(ids)
    # just return the most recent 3 stories:
    return ids[0:3]

ids = get_new_stories()

def get_story_data(id, anAgent={'User-Agent': 'PythonAiBook/1.0'}):
    req = Request(f"https://hacker-news.firebaseio.com/v0/item/{id}.json",
                  headers=anAgent)
    httpResponse = urlopen(req)
    return json.loads(httpResponse.read())

```

```python
def get_story_text(a_uri, anAgent={'User-Agent': 'PythonAiBook/1.0'}):
    req = Request(a_uri, headers=anAgent)
    httpResponse = urlopen(req)
    soup = BeautifulSoup(httpResponse.read(), "html.parser")
    return soup.get_text()

def find_entities_in_text(some_text):
    def clean(s):
        return s.replace('\n', ' ').strip()
    doc = spacy_model(some_text)
    return map(list, [[clean(entity.text), entity.label_] for entity in doc.ents])

import re
regex = re.compile('[^a-z_]')

def safe_prolog_text(s):
    s = s.lower().replace(' ','_').replace('&','and').replace('-','_')
    return regex.sub('', s)

for id in ids:
    story_json_data = get_story_data(id)
    #pprint(story_json_data)
    if story_json_data != None and 'url' in story_json_data:
        print(f"Processing {story_json_data['url']}\n")
        story_text = get_story_text(story_json_data['url'])
        entities = list(find_entities_in_text(story_text))
        #print(entities)
        organizations =
            set([safe_prolog_text(name)
                 for [name, entity_type] in entities if entity_type == "ORG"])
        people =
            set([safe_prolog_text(name)
                 for [name, entity_type] in entities if entity_type == "PERSON"])
        with PrologMQI() as mqi:
            with mqi.create_thread() as prolog_thread:
                for person in people:
                    s = f"assertz(person({person},
                                   '{story_json_data['url']}'))."
                    #print(s)
                    try:
                        prolog_thread.query(s)
                    except:
                        print(f"Error with term: {s}")
```

```
                for organization in organizations:
                    s = f"assertz(organization({organization}, '{story_json_data['ur\
l']}'))."
                    #print(s)
                    try:
                        prolog_thread.query(s)
                    except:
                        print(f"Error with term: {s}")
                try:
                    result = prolog_thread.query("organization(Organization, URI).")
                    pprint(result)
                except:
                    print("No results for organizations.")
                try:
                    result = prolog_thread.query("person(Person, URI).")
                    pprint(result)
                except:
                    print("No results for people.")
```

Here is some example output:

```
{'Organization': 'bath_and_beyond_inc',
  'URI': 'https://bedbathandbeyond.gcs-web.com/news-releases/news-release-details/be\
d-bath-beyond-inc-provides-business-update'},
{'Person': 'ryan_holiday',
  'URI': 'https://www.parttimetech.io/p/what-do-you-really-want'},
{'Organization': 'nasa_satellite',
 {'Organization': 'nih',
  'URI': 'https://nap.nationalacademies.org/catalog/26424/measuring-sex-gender-ident\
ity-and-sexual-orientation'},
  'URI': 'https://landsat.gsfc.nasa.gov/article/now-then-landsat-u-s-mosaics/'},
{'Organization': 'the_national_academies',
'URI': 'https://nap.nationalacademies.org/catalog/26424/measuring-sex-gender-identit\
y-and-sexual-orientation'},
{'Organization': 'microsoft',
  'URI': 'https://invention.si.edu/susan-kare-iconic-designer'},
{'Person': 'susan_kare',
  'URI': 'https://invention.si.edu/susan-kare-iconic-designer'},
```

Later we will use deep learning models to summarize text and other NLP tasks. This example could be extended for defining Prolog terms in the form ***summary(URI, "…")***. Other application ideas might be to use Python scripts for:

- Collect stock market data for a rule-based Prolog reasoner for stock purchase selections.
- A customer service chatbot that is mostly written in Python could be extended by using a Prolog based reasoning system.

Soar Cognitive Architecture

Soar* is a flexible and general purpose reasoning and knowledge management system for building intelligent software agents. The Soar project is a classic AI tool and has the advantage of being kept up to date. As I write this the Soar GitHub repository† was just updated a few days ago. Here is a bit of history:

Soar is a cognitive architecture that was originally developed at Carnegie Mellon University. It is a rule-based system that simulates the problem-solving abilities of human cognition. The architecture is composed of several interacting components, including a production system (a rule-based system), episodic memory (remembers past experiences), and working memory (for current state of the world).

The production system is the core component of Soar and is responsible for making decisions and performing actions. It is based on the idea of a production rule, which is a rule that describes a condition and an action to be taken when that condition is met. Production systems use rules to make decisions and to perform actions based on the current state of the system.

The episodic memory is a component that stores information about past events and experiences. It allows a system built with Soar to remember past events and use that information to make decisions affecting the future state of the world. The working memory is a component that stores information about the current state of the system and is used by the production system to make decisions.

Soar also includes several other components, such as an explanation facility, which allows the system to explain its decisions and actions, and a learning facility, which allows the system to learn from its experiences.

Soar is a general-purpose architecture that can be applied to a wide range of tasks and domains. In the past it has been particularly well-suited for tasks that involve planning, problem-solving, and decision-making. It has been used in a variety of applications, including robotics, NLP, and intelligent tutoring systems.

I am writing this material many years after my original use of the Soar project. My primary reference for preparing the following material is the short paper Introduction to the Soar Cognitive Architecture‡ by John E. Laird. For self-study you can start at the Soar Tutorial Page§ that provides a download for an eight part tutorial in separate PDF files, binary Soar executables for Linux, Windows, and macOS, and all of the code for the tutorials.

*https://soar.eecs.umich.edu/
†https://github.com/SoarGroup/Soar
‡https://arxiv.org/pdf/2205.03854.pdf
§https://soar.eecs.umich.edu/articles/downloads/soar-suite/228-soar-tutorial-9-6-0

I consider Soar to be important because it proposes and implements a general purpose cognitive architecture. A warning to the reader: Soar has a steep learning curve and there are simpler frameworks for solving practical problems. Later we will look at an example from the Soar Tutorial for the classic "blocks world" problem of moving blocks on a table subject to constraints like not being allowed to move a block if it has another object on top of it. Solving this fairly simple problem requires about 400 lines of Soar source code.

Background Theory

The design goals for the Soar Cognitive Architecture (which I will usually refer to as Soar) is to provide "fixed structures, mechanisms, and representations"* to develop human level behavior across a wide range of tasks. There is a commercial company Soar.com† that uses Soar for commercial and government projects.

We will cover Reinforcement Learning‡ (which I will usually refer to as RL) in a later chapter but there is similar infrastructure supported by both Soar and RL: a simulated environment, data representing the state of the environment, and possible actions that can be performed in the environment that change the state.

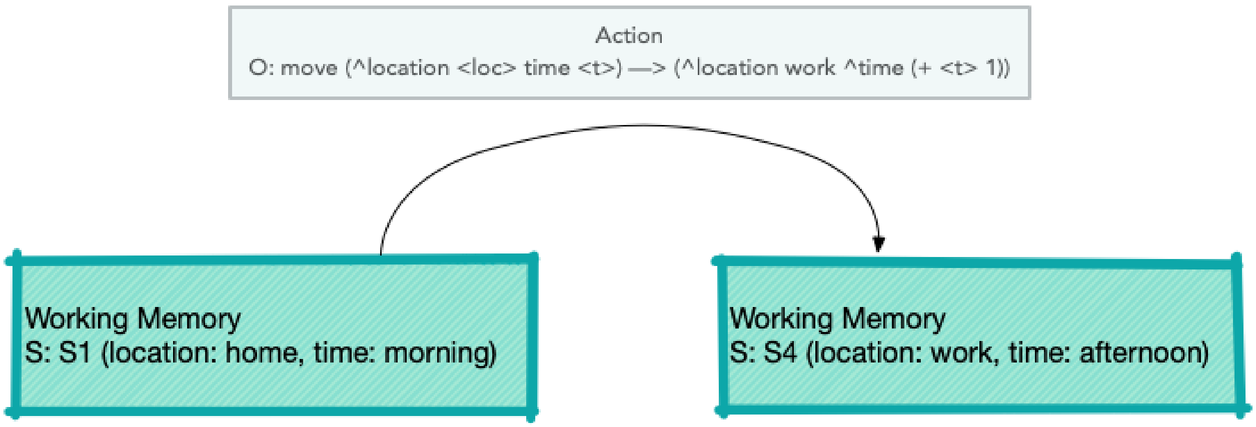
As we have disused, there are two main types of memory of the Soar architecture:

- Working Memory - this is the data that specifies the current state of the environment. Actions in the environment change the data in working memory, either by modification, addition, or deletion. At the risk of over-anthropomorphism, consider this like human short term memory.
- Production Memory - this data is a form of production rules where the left-hand side of rules consist of patterns that if matched against working memory, the the actions on the right-hand side of a rule are executed. Consider these production rules as long-term memory.

Both Soar working memory and production memory are symbolic data. As a contrast, data in deep learning is numeric, mostly tensors. This symbolic data comprises goals (G), problem spaces (PS), states (S) and operators (O).

Soar Operator transitioning from one state to another (figure is from the Soar Tutorial):

*https://soar.eecs.umich.edu/workshop/30/laird2.pdf
†https://try.soar.com
‡https://en.wikipedia.org/wiki/Reinforcement_learning



Setup Python and Soar Development Environment

It will take you a few minutes to install Soar on your system and create the Python bindings. Start by cloning the Soar GitHub repository[*] and run the install script from the top directory:

```
python scons/scons.py sml_python
```

If you want all language bindings replace **sml_python** with **all**. Change directory to the **out** subdirectory and note the directory path. On my system:

```
$ pwd
/Users/markw/SOAR/Soar/out
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/Users/markw/SOAR/Soar/out
$ export PYTHONPATH=$PYTHONPATH:/Users/markw/SOAR/Soar/out
```

I will present here a simple example and explain a subset of the capabilities of Soar. When we are done here you can reference a recent paper by Neha Rajan and 2Sunderrajan Srinivasan Exploring Learning Capability of an Agent in SOAR: Using 8- Queens Problem[†] for a complete example using Soar for cognitive modeling and a more complex example.

A minimal Soar Tutorial

I am presenting a minimal introduction to Soar and we will later provide an example of Python and Soar interop for the purpose of introducing you to Soar. If this material looks interesting then I encourage you to work through the Soar Tutorial Page[‡].

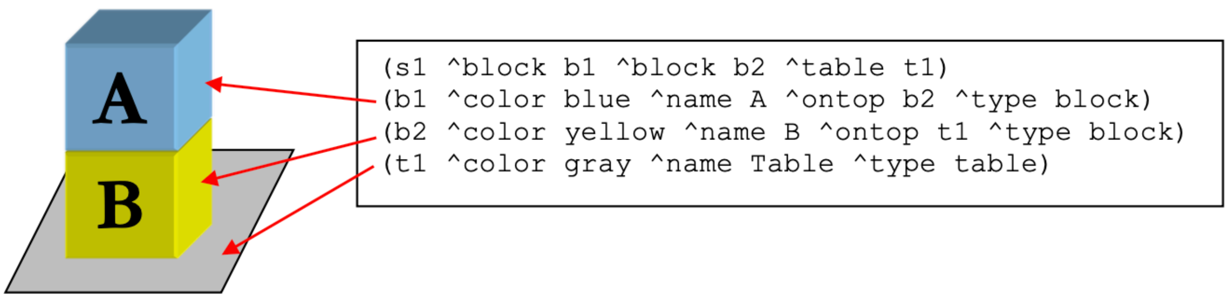
Soar supports a rule language that uses the highly efficient Rete algorithm[§] (optimized for huge numbers of rules, less optimized for large working memories). Let's look at a sample rule from Chapter 1 (first PDF file) of the Soar tutorial:

[*] https://github.com/SoarGroup/Soar
[†] https://thescipub.com/pdf/jcssp.2020.642.650.pdf
[‡] https://soar.eecs.umich.edu/articles/downloads/soar-suite/228-soar-tutorial-9-6-0
[§] https://en.wikipedia.org/wiki/Rete_algorithm

```
1 sp {hello-world
2    (state <s> ^type state)
3 -->
4    (write |Hello World|)
5    (halt)
6 }
```

The token **sp** on line 1 stands for Soar Production. Rules are enclosed in **{** and **}**. The name of this rule is the symbol **hello-world**. In the tutorial you will usually see rule names partitioned using the characters * and -. Rules have a “left side” and a “right side”, separated by —>. If all of the left side patterns match working memory elements then the right-hand side actions are executed.

The following figure is from the Soar tutorial and shows two blocks stacked on top of each other. The bottom block rests on a table:



This figure represents state **s1** that is a root of the graph also containing blocks named **b1** and **b2** as well as the table named **t1**. The blocks and table all have attributes ^**color**, ^**name**, and ^**type**. The blocks also have the optional attribute ^**ontop**.

Rule right-hand side actions can modify, delete, or add working memory data. For example, a left-hand side matching the attribute values for block **b1** could modify its ^**ontop** attribute from the value **b2** to the table named **t1**.

Example Soar System With Python Interop

We will use the simplest blocks world example in the Soar Tutorial in our Python interop example. In the examples directories in the Soar Tutorial, this example is spread through eight source files. I have copied them to a single file **Soar/blocks-world/bw.soar** in the GitHub repository for this book.

```python
import Python_sml_ClientInterface as sml

def callback_debug(mid, user_data, agent, message):
    print(message)

if __name__ == "__main__":
    soar_kernel = sml.Kernel.CreateKernelInCurrentThread()
    soar_agent = soar_kernel.CreateAgent("agent")
    soar_agent.RegisterForPrintEvent(sml.smlEVENT_PRINT, callback_debug, None) # no \
user data
    soar_agent.ExecuteCommandLine("source bw.soar")
    run_result=soar_agent.RunSelf(50)
    soar_kernel.DestroyAgent(soar_agent)
    soar_kernel.Shutdown()
```

Run this example:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/Users/markw/SOAR/Soar/out
$ export PYTHONPATH=$PYTHONPATH:/Users/markw/SOAR/Soar/out
$ python bw.py

     1:    O: O1 (initialize-blocks-world)
Five Blocks World - just move blocks.
The goal is to get EDBCA.
AC
DEB

     2:    O: O8 (move-block)
 Apply O8: move-block(C,table)P10*apply*move-block*internal
A
DEB
C

     3:    O: O7 (move-block)
 Apply O7: move-block(B,table)P10*apply*move-block*internal
DE
B
C
A

     4:    O: O17 (move-block)
 Apply O17: move-block(E,table)P10*apply*move-block*internal
D
```

```
B
C
A
E

     5:    O: O25 (move-block)
 Apply O25: move-block(D,E)P10*apply*move-block*internal
B
C
A
ED

     6:    O: O27 (move-block)
 Apply O27: move-block(C,D)P10*apply*move-block*internal
B
EDC
A

     7:    O: O10 (move-block)
 Apply O10: move-block(B,C)P10*apply*move-block*internal
EDCB
A

     8:    O: O11 (move-block)
 Apply O11: move-block(A,B)P10*apply*move-block*internal
EDCBA
Goal Achieved (five blocks).
System halted.
Interrupt received.This Agent halted.
```

I consider Soar to be of historic interest and is an important example of a large multiple-decade research project in building large scale reasoning systems.

Constraint Programming with MiniZinc and Python

As with Soar, our excursion into constraint programming will be brief, hopefully enough to introduce you to a new style of programming though a few examples. I still use constraint programming and hope you might find the material in this section useful.

While I attempt to make this material self-contained reading, you may want to use the MiniZinc Python* documentation as a reference for the Python interface and The MiniZinc Handbook† as a

*https://minizinc-python.readthedocs.io/en/latest/getting_started.html
†https://www.minizinc.org/doc-2.6.4/en/index.html

reference to the MiniZinc language and its use for your own projects.

Constraint Programming (CP) is a paradigm of problem-solving that involves specifying the constraints of a problem, and then searching for solutions that satisfy those constraints. MiniZinc is a high-level modeling language for constraint programming that allows users to specify constraints and objectives in a compact and expressive way.

In MiniZinc a model is defined by a set of variables and a set of constraints that restrict the possible values of those variables. The variables can be of different types, such as integers, booleans, and sets, and the constraints can be specified using a variety of built-in predicates and operators. The model can also include an objective function that the solver tries to optimize.

Once a model is defined in MiniZinc, it can be solved using a constraint solver which is a software program that takes the model as input and returns solutions that satisfy the constraints. MiniZinc supports several constraint solvers, including Gecode, Chuffed, and OR-Tools, each of which has its own strengths and weaknesses.

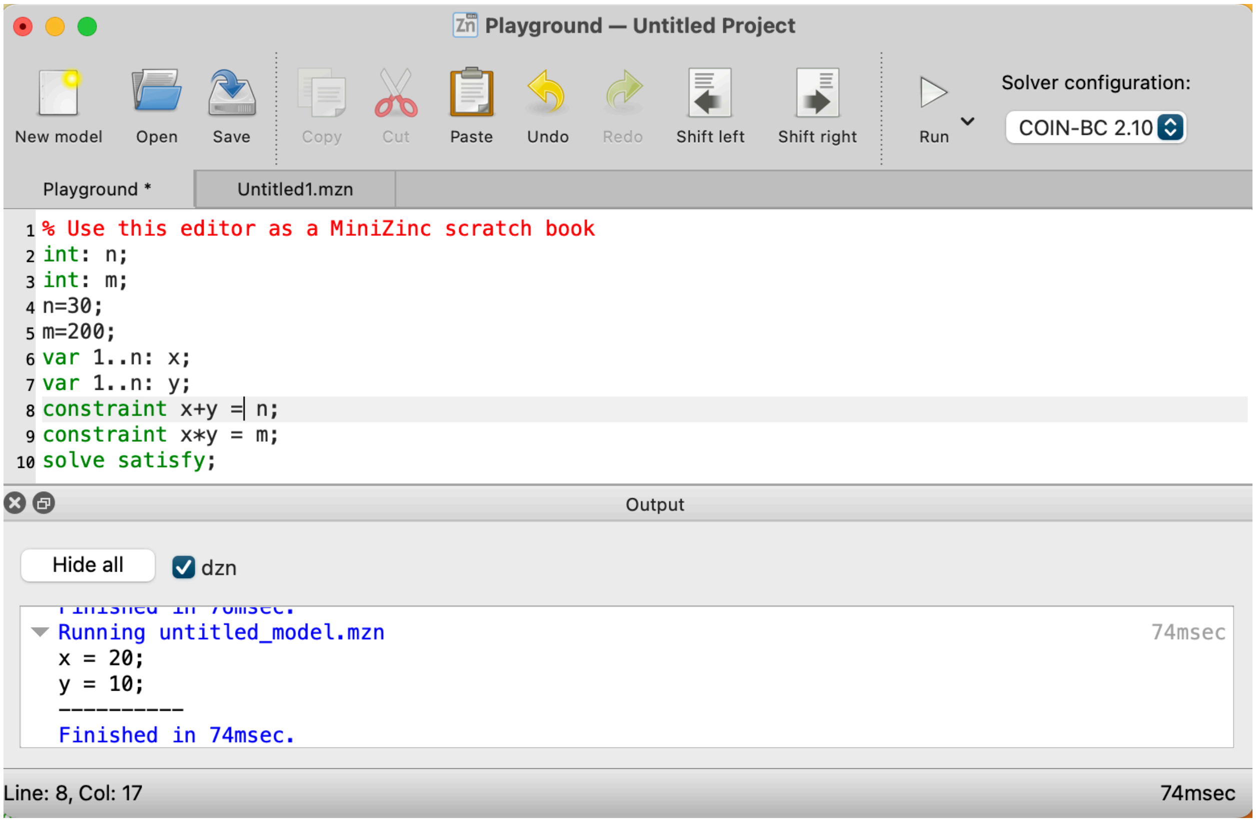
MiniZinc provides a feature called "Annotations", which allows the user to specify additional information to the solver, as the way to search for solutions, or setting the maximum time for the solver to run.

Installation and Setup for MiniZinc and Python

You need to first install the MiniZinc system. For macOS this can be done with **brew install minizinc** or can be installed from source code on macOs and Linux*. The Python interface can be installed with **pip install minizinc**.

The following figure shows the MiniZincIDE with simple constraint satisfaction problem:

*https://www.minizinc.org/doc-2.5.5/en/installation_detailed_linux.html



When I installed **minizinc** on macOS with **brew**, the solver **coinbc** was installed automatically so that is what we use here. Here is the MiniZinc source file **test1.mzn** that you also see in the last figure:

```
int: n;
int: m;
var 1..n: x;
var 1..n: y;
constraint x+y = n;
constraint x*y = m;
```

There are several possible solvers to use with MiniZinc. When I install on macOS using *brew* the solver "coinbc" is available. When I install **sudo apt install minizinc** on Ubuntu Linux, the solver "gecode" is available.

Notice that we don't set values for the constants **n** and **m** as we did when using MiniZincIDE. We instead set them in Python code before calling the solver in lines 7 and 8:

```
from minizinc import Instance, Model, Solver

coinbc = Solver.lookup("coinbc")

test1 = Model("./test1.mzn")
instance = Instance(coinbc, test1)
instance["n"] = 30
instance["m"] = 200

result = instance.solve()
print(result)
print(result["x"])
print(result["y"])
```

The result is:

```
$ python test1.py
Solution(x=20, y=10, _checker='')
20
10
```

Let's look at a more complex example: on the map of the USA, the states neighboring each other are colored differently than their adjoining states in such a way that no states with touching edges are the same color. We use integers to represent colors and the mapping of numbers to colors is unimportant. Here is a partial listing of **us_states.mzn**:

```
int: nc = 3; %% needs to be 4 to solve this problem

var 1..nc: alabama;
var 1..nc: alaska;
var 1..nc: arizona;
var 1..nc: arkansas;
var 1..nc: california;
 ...
constraint alabama != florida;
constraint alabama != georgia;
constraint alabama != mississippi;
constraint alabama != tennessee;
constraint arizona != california;
constraint arizona != colorado;
constraint arizona != nevada;
constraint arizona != new_mexico;
```

```
17 constraint arizona != utah;
18  ...
19 solve satisfy;
```

In line 9, as an example, the statement **constraint alabama != florida;** means that since Alabama and Florida share a border, they must have different color indices. Initially we set the number of allowed color to 3 on line 1 and with just three colors allowed this problem is unsolvable.

The output is:

```
 $ minizinc --solver coinbc us_states.mzn
=====UNSATISFIABLE=====
```

So we need more than three colors. Let's try **int: nc = 4;**:

```
$ minizinc --solver coinbc us_states.mzn
alabama = 2;
alaska = 1;
arizona = 3;
arkansas = 4;
california = 4;
colorado = 4;
connecticut = 2;
delaware = 4;
 ...
```

Here is a Python script **us_states.py** that uses this model and picks out the assigned color indices from the solution object:

```
from minizinc import Instance, Model, Solver

coinbc = Solver.lookup("coinbc")

model = Model("./us_states.mzn")
instance = Instance(coinbc, model)
instance["nc"] = 4 # solve for a maximum of 4 colors

result = instance.solve()
print(result)
all_states = list(result.__dict__['solution'].__dict__.keys())
all_states.remove('_checker')
print(all_states)
for state in all_states:
    print(f" {state} \t: \t{result[state]}")
```

Here is some of the output:

```
$ python us_states.py
Solution(alabama=2, alaska=1, arizona=3, arkansas=4, california=4, colorado=4, conne\
cticut=2, delaware=4, florida=1, georgia=4, hawaii=1, idaho=4, ... ]
 alabama    :   2
 alaska     :   1
 arizona    :   3
 arkansas   :   4
 ...
 wisconsin  :   1
 wyoming    :   3
```

Good Old Fashioned Symbolic AI Wrap-up

As a practical matter almost all of my work in the last ten years used either deep learning or was comprised of a combination of semantic web and linked data with deep learning projects. While the material in this chapter is optional for the modern AI practitioner, I still find using MiniZinc for constraint programming and Prolog to be useful. I included the material for the Soar cognitive architecture because I both find it interesting and I believe the any future development of “real AI” (or AGI) will involve hybrid approaches and there are many good ideas in the Soar implementation.

Part II - Knowledge Representation

In the next two chapters we will look at a variety of techniques for encoding information and knowledge for automated processing by AI systems. We will follow the material from the last chapter by diving into using graph data stores and relational databases. We finish this part of the book by reviewing the theory behind the Semantic Web and Linked Data and look at some practical applications.

We touched on symbolic knowledge representation in the last chapter. Here and in the next two chapters we cover organizing and using "knowledge as data" using graph and relational data stores. Generally in software development we start any project by evaluating sources of data, format and schemas, and what operations we need to accomplish.

AI development is similar but different. Often we need to infer data that is missing, reason about uncertainty, and generally organize data to capture features of changing environments, etc. In a later chapter on the semantic web and linked data we will see formalisms to infer missing data based on Ontological rules.

Getting Setup To Use Graph and Relational Databases

I use several types of data stores in my work but for the purposes of this book generally and for this chapter specifically we can explore interesting ideas and lay a foundation for examples later in this book for using graph and relational databases using just three platforms:

- Apache Jena Fuseki* for RDF data storage, SPARQL queries†, and Fuseki's web interface to explore datasets.
- Neo4j Community Edition‡ for a transactional disk-based graph database, the Cypher query language§, and the web interface to explore datasets. If you prefer you can alternatively use Memgraph¶ that is fairly compatible with Neo4j.
- SQLite‖ for transactional relational data storage and the SQL query language**.

The next chapter covers RDF and the SPARQL query language in some detail.

In technical terms, knowledge representation using graph and relational databases involves the use of graph structures and relational data models to represent and organize knowledge in a structured, computationally efficient, and easily accessible way.

A graph structure is a collection of nodes (also known as vertices) and edges (also known as arcs) that connect the nodes. Each node and edge in a graph can have properties, such as labels and attributes which provide information about the entities they represent. Graphs can be used to represent knowledge in a variety of ways, such as through semantic networks and using ontologies to define terms, classes, types, etc.

Relational databases, on the other hand, use a tabular data model to represent knowledge. The basic building block of a relational database is the table, which is a collection of rows (also known as tuples) and columns (also known as attributes). Each row represents an instance of an entity, and the columns provide information about the properties of that entity. Relationships between entities can also be represented by foreign keys, which link one table to another.

Combining these two technologies, knowledge can be represented as a graph of interconnected entities, where each entity is stored in a relational database table and connected to other entities through relationships represented by edges in the graph. This allows for efficient querying and

*https://jena.apache.org/documentation/fuseki2/
†https://jena.apache.org/tutorials/sparql.html
‡https://neo4j.com/download-center/#community
§https://neo4j.com/docs/cypher-manual/current/
¶https://memgraph.com
‖https://www.sqlite.org/index.html
**https://en.wikipedia.org/wiki/SQL

manipulation of knowledge, as well as the ability to integrate and reason over large amounts of information.

The Apache Jena Fuseki RDF Datastore and SPARQL Query Server

I use several RDF datastores in my work (Franz AllegroGraph, Stardog, GraphDB, and Blazegraph) but I particularly like Apache Jena Fuseki (that I will often just call Fuseki) because it is open source, it follows new technology like RDF* and SPARQL*, and has a simple and easy to use web interface). Most of our experiments will use Python SPARQL client libraries but you will also be spending quality time using the web interface to run SPARQL queries.

RDF data is in the form of triples: subject, property (or predicate), and object. There are several serialization formats for RDF including XML, Turtle, N1, etc. I refer you to the chapter Background Material for the Semantic Web and Knowledge Graphs in my book Practical Artificial Intelligence Programming in Clojure* for more details (link for online reading). Here we use client libraries and web services that can return RDF data as JSON.

This chapter is about getting tools ready to use. In the next chapter we will get more into RDF and SPARQL.

I have a GitHub repository containing Fuseki, sample data, and directions for getting setup and running in a minute or two: https://github.com/mark-watson/fuseki-semantic-web-dev-setup. You can clone this repository and follow along on your laptop or just read the following text if you are not yet convinced that you will want to use semantic web technologies in your own projects.

We will be using the SPARQL query language here for a few examples and then jump more deeply into the use of SPARQL and other semantic web technologies in the next chapter.

Listing of **test_fuseki_client.py** (in the directory **semantic-web**):

```
## Test client for Apache Jena Fuselki server on localhost
##
## Do git clone https://github.com/mark-watson/fuseki-semantic-web-dev-setup
## and run ./fuseki-server --file RDF/fromdbpedia.ttl /news
## in the cloned directory before running this example.

import rdflib
from SPARQLWrapper import SPARQLWrapper, JSON
from pprint import pprint

queryString = """
SELECT *
```

*https://markwatson.com/books/clojureai-site/#semantic-web

```
WHERE {
    ?s ?p ?o
}
LIMIT 5000
"""

sparql = SPARQLWrapper("http://localhost:3030/news")
sparql.setQuery(queryString)
sparql.setReturnFormat(JSON)
sparql.setMethod('POST')
ret = sparql.queryAndConvert()
for r in ret["results"]["bindings"]:
    pprint(r)
```

This generates output (edited for brevity):

```
$ python test_fuseki_client.py
{'o': {'datatype': 'http://www.w3.org/2001/XMLSchema#decimal',
       'type': 'literal',
       'value': '56.5'},
 'p': {'type': 'uri', 'value': 'http://dbpedia.org/property/janHighF'},
 's': {'type': 'uri', 'value': 'http://dbpedia.org/resource/Sedona,_Arizona'}}
{'o': {'datatype': 'http://www.w3.org/2001/XMLSchema#integer',
       'type': 'literal',
       'value': '0'},
 'p': {'type': 'uri', 'value': 'http://dbpedia.org/property/yearRecordLowF'},
 's': {'type': 'uri', 'value': 'http://dbpedia.org/resource/Sedona,_Arizona'}}
{'o': {'datatype': 'http://www.w3.org/2001/XMLSchema#decimal',
       'type': 'literal',
       'value': '5.7000000000000001776'},
 'p': {'type': 'uri',
       'value': 'http://dbpedia.org/property/sepPrecipitationDays'},
 's': {'type': 'uri', 'value': 'http://dbpedia.org/resource/Sedona,_Arizona'}}
```

When I use RDF data from public SPARQL endpoints like DBPedia or Wikidata in applications I start by using the web based SPARQL clients for these services, find useful entities, manually look to see what properties (or predicates) and defined for these entities, and then write custom SPARQL queries to fetch the data I need for any specific application. In a later chapter we will build a Python command line tool to partially automate this process.

The following example **fuseki_cities_and_coldest_temperatures.py** finds 20 DBPedia URIs of the type "city" (the tiny part of DBPedia that we have loaded in our local Fuseki instance the only city is home town of Sedona Arizona) and collects data for a few properties for each city URI. Later we will make the same query against the DBPedia public SPARQL endpoint.

```python
import rdflib
from SPARQLWrapper import SPARQLWrapper, JSON
from pprint import pprint

queryString = """
SELECT *
WHERE {
    ?city_uri
        <http://dbpedia.org/ontology/type>
        <http://dbpedia.org/resource/City> .
    ?city_uri
        <http://dbpedia.org/property/yearRecordLowF>
        ?record_year_low_temperature .
    ?city_uri
        <http://dbpedia.org/property/populationEst>
        ?population .
    ?city_uri
         <http://www.w3.org/2000/01/rdf-schema#label>
         ?dbpedia_label FILTER (lang(?dbpedia_label) = 'en') .
}
LIMIT 20
"""

sparql = SPARQLWrapper("http://localhost:3030/news")
sparql.setQuery(queryString)
sparql.setReturnFormat(JSON)
sparql.setMethod('POST')
ret = sparql.queryAndConvert()
for r in ret["results"]["bindings"]:
    pprint(r)
```

The output is:

```
$ python fuseki_cites_and_coldest_temperatures.py
{'city_uri': {'type': 'uri',
              'value': 'http://dbpedia.org/resource/Sedona,_Arizona'},
 'dbpedia_label': {'type': 'literal',
                   'value': 'Sedona, Arizona',
                   'xml:lang': 'en'},
 'population': {'datatype': 'http://www.w3.org/2001/XMLSchema#integer',
                'type': 'literal',
                'value': '10281'},
 'record_year_low_temperature': {'datatype':
```

```
                                 'http://www.w3.org/2001/XMLSchema#integer',
                                 'type': 'literal',
                                 'value': '0'}}
```

We can make a slight change to this example to access the public DBPedia SPARQL endpoint instead of our local Fuseki instance. I copied the above Python script, renamed it to **dbpedia_cities_and_coldest_temperatures.py** changing only the URI of the SPARQL endpoint and commented setting the HTTP method to POST:

```
sparql = SPARQLWrapper("http://dbpedia.org/sparql")
#sparql.setMethod('POST')
```

The output when querying the public DBPedia SPARQL endpoint is (output edited for brevity, showing only two cities out of the thousands in DBPedia):

```
$ python dbpedia_cities_and_coldest_temperatures.py
{'city_uri': {'type': 'uri',
              'value': 'http://dbpedia.org/resource/Allardt,_Tennessee'},
 'dbpedia_label': {'type': 'literal',
                   'value': 'Allardt, Tennessee',
                   'xml:lang': 'en'},
 'population': {'datatype': 'http://www.w3.org/2001/XMLSchema#integer',
                'type': 'typed-literal',
                'value': '627'},
 'record_year_low_temperature': {'datatype':
                                 'http://www.w3.org/2001/XMLSchema#integer',
                                 'type': 'typed-literal',
                                 'value': '-27'}}
{'city_uri': {'type': 'uri',
              'value': 'http://dbpedia.org/resource/Dayton,_Ohio'},
 'dbpedia_label': {'type': 'literal',
                   'value': 'Dayton, Ohio',
                   'xml:lang': 'en'},
 'population': {'datatype': 'http://www.w3.org/2001/XMLSchema#integer',
                'type': 'typed-literal',
                'value': '137644'},
 'record_year_low_temperature': {'datatype':
                                 'http://www.w3.org/2001/XMLSchema#integer',
                                 'type': 'typed-literal',
                                 'value': '-28'}}
 ...
```

We will use more SPARQL queries in the next chapter.

The Neo4j Community Edition and Cypher Query Server and the Memgraph Graph Database

The examples here use Neo4j. You can either install Neo4J using these instructions* and follow along or just read the sample code and the sample output if you are not sure will be using property graph databases in your applications.

Note: You can alternatively use the Memgraph Graph Database† that is largely compatible with Neo4j. However, we are using a sample graph database that is included with free community edition of Neo4J so you will need to do extra work following this material using Memgraph.

We will use a Python client to access the Neo4J sample Movie Data graph database.

Admin Tricks and Examples for Using Neo4j

While reading this section please open the web page for the Neo4J Cypher query language tutorial‡ for reference since I will not duplicate that information here.

The Python source code for this section can be found in the directory **neo4j**.

When writing Python scripts to create data in Neo4J it is useful to remove all data from a graph and to verify removal:

```
MATCH (n) DETACH DELETE n
MATCH (n) RETURN n
```

We will use the interactive Movie database tutorial data that is built into the Community Edition of Neo4j. I assume that you have the tutorial running on a local copy of Neo4J or have the Cypher tutorial open. The following Cypher snippet creates a movie graph node with properties *title*, *released* year, and *tagline* for the movie The Matrix. Two nodes are created for actors Keanu Reeves and Carrie-Anne Moss that have properties *name* and *born* (for their birth year). Finally we create two links indicating the both actors stared in the move The Matrix.

*https://neo4j.com/docs/operations-manual/current/installation/
†https://memgraph.com
‡https://neo4j.com/developer/cypher/guide-cypher-basics/

```
CREATE (TheMatrix:Movie
         {title:'The Matrix', released:1999,
          tagline:'Welcome to the Real World'})
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
CREATE (Carrie:Person {name:'Carrie-Anne Moss', born:1967})
CREATE
      (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix),
      (Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrix))
```

Of particular use and interest is the ability to also define properties for links between nodes. If you use a property graph in your application you will start by:

- Document the types of nodes and links you will require for your application and what properties will be attached to the types of nodes and links.
- Write a Python load script to convert your data sources to Cypher **CREATE** statements and populate your Neo4J database.
- Write Python utilities for searching, modifying, and removing data.
- Write your Python application.

Let's continue with the Cypher tutorial material by adding data constraints to ensure that all movie and actor node names are unique:

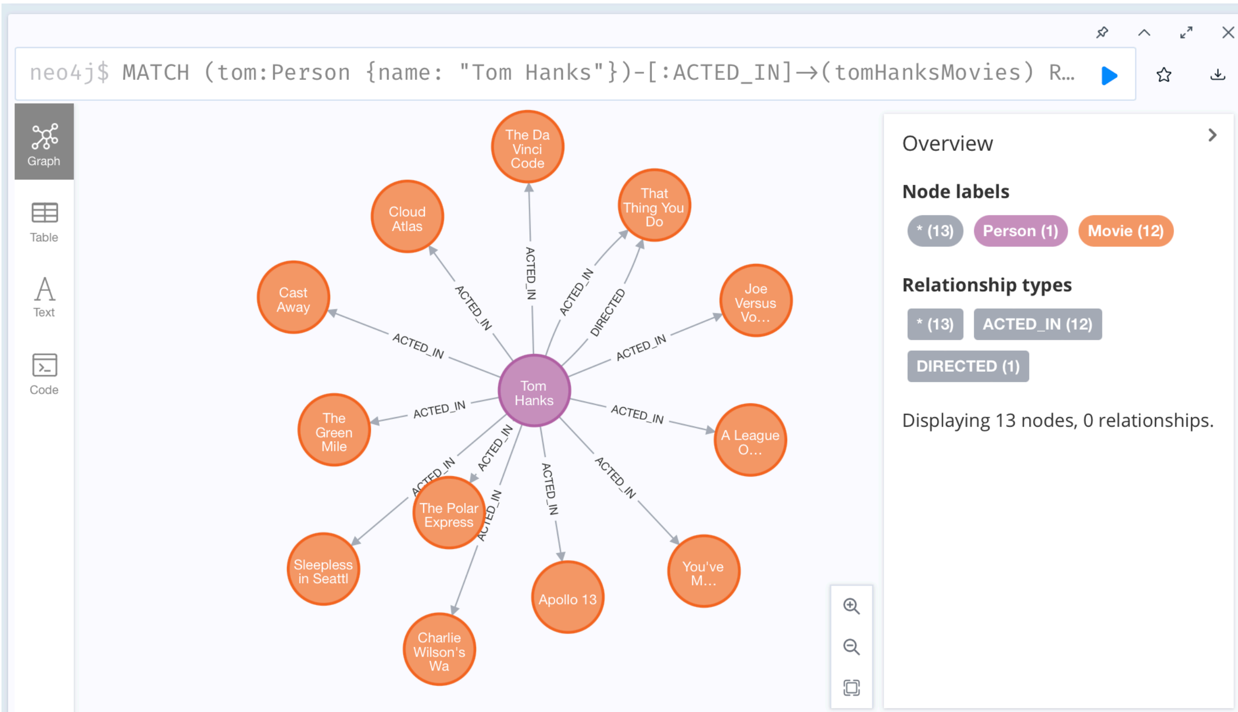
```
CREATE CONSTRAINT FOR (n:Movie) REQUIRE (n.title) IS UNIQUE
CREATE CONSTRAINT FOR (n:Person) REQUIRE (n.name) IS UNIQUE
```

Index all movie nodes on the key of movie release date:

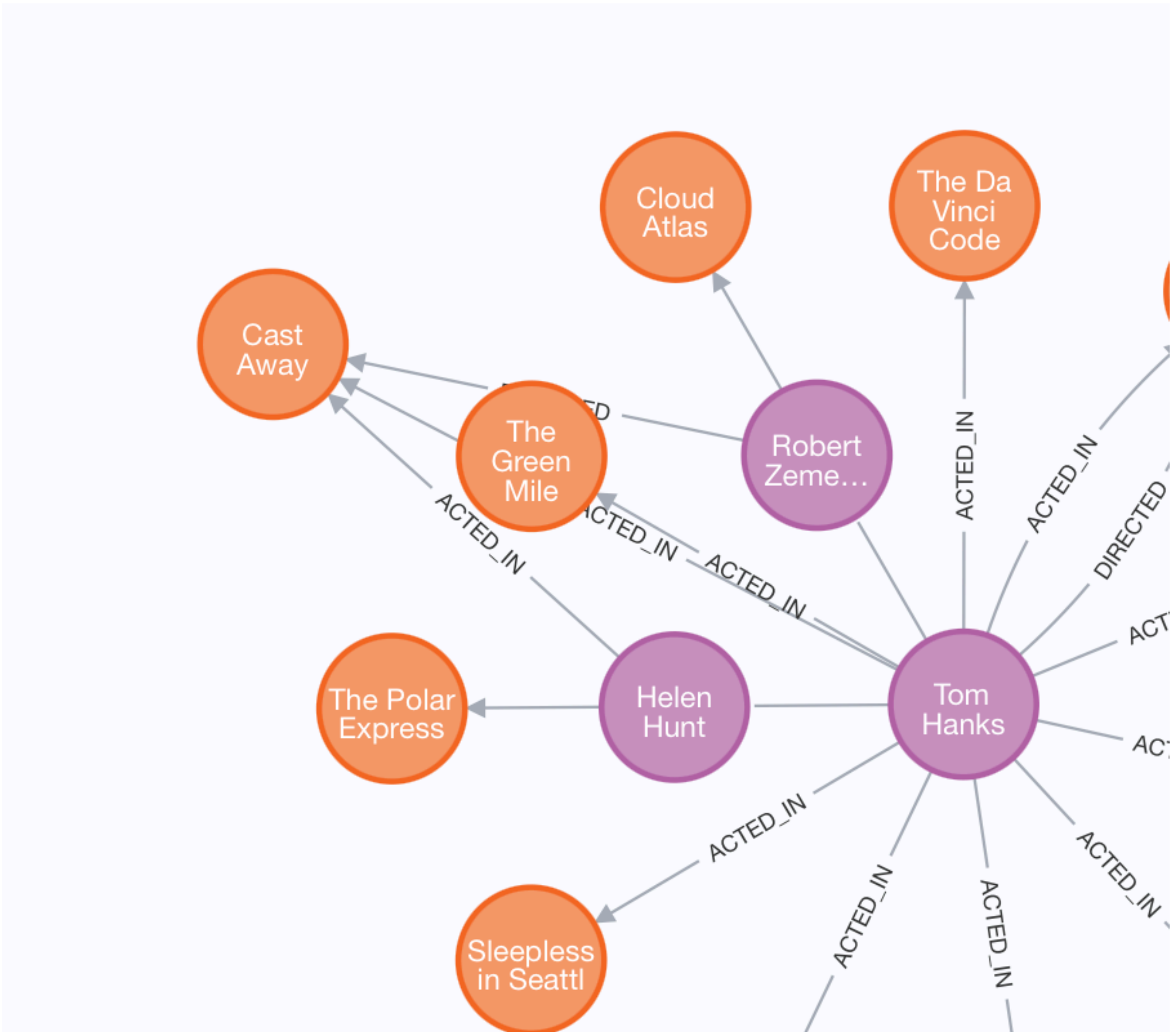
```
CREATE INDEX FOR (m:Movie) ON (m.released)
```

This is not strictly necessary but just as we index columns in a relational database, indexing nodes can drastically speed up queries. Here is a test query from the Neo4J tutorial:

```
MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->(tomHanksMovies)
RETURN tom,tomHanksMovies
```



By default, not all links are shown. If we double click on the node "Cast Away" in the upper left corner then all links from that node are shown in the display graph:



We can query for movies created during a specific time period:

```
MATCH (nineties:Movie) WHERE nineties.released >= 1990 AND
                             nineties.released < 2000
RETURN nineties.title
```

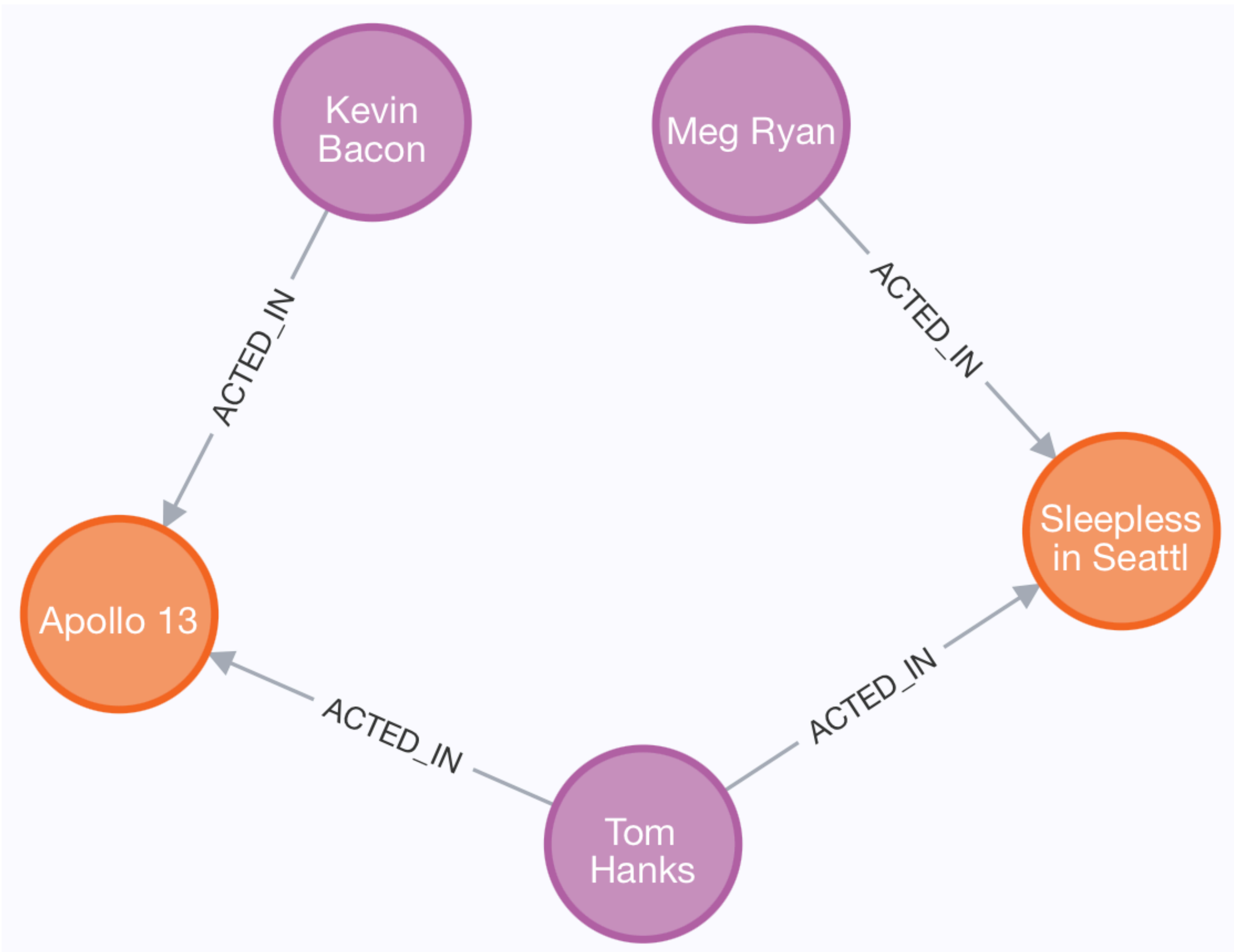
The following query returns the names of all actors who co-starred with Tom Hanks in any movie:

```
MATCH (tom:Person
        {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors)
RETURN DISTINCT coActors.name
```

Here we use a built in function to find the shortest path in the movie data graph between Kevin Bacon and Meg Ryan. The Cypher graph expression **-[*]-** represnts any path between the “Kevin Bacon” and “Meg Ryan” nodes in the graph:

```
MATCH p=shortestPath(
    (bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person {name:"Meg Ryan"})
)
RETURN p
```

Here is the shortest path:



Python client code for the Neo4J Movie graph database example

The following listing shows an example from the Neo4J documentation that I modified:

```python
import logging, sys, os

from neo4j import GraphDatabase
from neo4j.exceptions import ServiceUnavailable

USER = 'neo4j'
PASSWORD = os.environ.get('NEO4J_AURADB_PASSWORD')

class MovieDbExample:
    "Boilerplate code copied from Neo4J Python client documentation"

    def __init__(self, uri, user, password):
        self.driver = GraphDatabase.driver(uri, auth=(user, password))

    def close(self):
        self.driver.close()

    @staticmethod
    def enable_log(level, output_stream):
        handler = logging.StreamHandler(output_stream)
        handler.setLevel(level)
        logging.getLogger("neo4j").addHandler(handler)
        logging.getLogger("neo4j").setLevel(level)

    def find_movies(self, actor_name):
        with self.driver.session() as session:
            result = session.execute_read(self._movies_actor_is_in, actor_name)
            for row in result:
                print("Movie: {row}".format(row=row))

    @staticmethod
    def _movies_actor_is_in(tx, actor_name):
        query = (
            "MATCH (actor:Person {name: $actor_name})-[:ACTED_IN]->(movies) "
            "RETURN movies.title as title"
        )
        result = tx.run(query, actor_name=actor_name)
        return [row["title"] for row in result]

if __name__ == "__main__":
    bolt_url = "neo4j://localhost:7687"
    MovieDbExample.enable_log(logging.INFO, sys.stdout)
    MovieDbExample = MovieDbExample(bolt_url, USER, PASSWORD)
```

```
    MovieDbExample.find_movies("Tom Hanks")
    MovieDbExample.close()
```

For practical applications, you will write many helper functions for executing required Cypher queries. Before you write one line of Python code I suggest that you always experiment in the Neo4J web app with test graph database data and interactively write the Cypher queries you need. Once you have working queries then write the Python client code based on both the example we just looked at and the Neo4J Python documentation.

This ends our brief tour of property graphs. In my own work I use semantic web RDF graph databases for most of my work so we will later take a much deeper dive into RDF and the SPARQL query language, but not further use property graphs in this book except for support both RDF and Cypher data generation in a later example Knowledge Graph Creator. I personally use RDF for about 90% of my work with graph data and property graphs like Neo4J about 10% of the time.

I wanted to introduce you to property graphs because I know developers who have an easier time using property graphs. I believe that it is worth your time, dear reader, to experiment a bit with both approaches and then choose a favorite

The SQLite Relational Database

The SQLite database is now included in the standard Python distribution so SQLite is my default persistent datastore. I tend to use RDF and SPARQL (or occasionally Neo4J) specifically when a graph database fits the requirements an application. The example code for this section can be found in the directory **/misc/datastores** that also includes examples for Postgres that I don't cover in the book text. We will also use SQLite in a later chapter in the Knowledge Graph Navigator example to cache SPARQL queries to DBPedia.

We start with writing a simple reusable library for SQLite using the standard library **sqlite3** that is defined in the file **sqlite_lib.py**:

```
from sqlite3 import connect, version

def create_db(db_file_path):
    "create database"
    conn = connect(db_file_path)
    print(version)
    return conn.close()

def connection(db_file_path):
    "create database connection"
    return connect(db_file_path)

```

```
def query(conn, sql, variable_bindings=None):
    "run a test query"
    cur = conn.cursor()
    status = cur.execute(sql, variable_bindings) if variable_bindings else cur.execu\
te(sql)
    print(f"query status: {status}")
    return cur.fetchall()
```

Here is a test program **sqlite_example.py**:

```
from sqlite_lib import create_db, connection, query

def test_sqlite_lib():
    "test library"
    dbpath = ':memory:'
    create_db(dbpath)
    conn = connection(':memory:')
    query(conn, 'CREATE TABLE people (name TEXT, email TEXT);')
    print(query(conn,
        "INSERT INTO people VALUES ('Mark', 'mark@markwatson.com')"))
    print(query(conn,
        "INSERT INTO people VALUES ('Kiddo', 'kiddo@markwatson.com')"))
    print(query(conn, 'SELECT * FROM people'))
    print(query(conn, 'UPDATE people SET name = ? WHERE email = ?', [
        'Mark Watson', 'mark@markwatson.com']))
    print(query(conn, 'SELECT * FROM people'))
    print(query(conn, 'DELETE FROM people  WHERE name=?', ['Kiddo']))
    print(query(conn, 'SELECT * FROM people'))
    return conn.close()

test_sqlite_lib()
```

We will combine the use of SQLite, RDF and SPARQL, and deep learning Natural Language Processing (NLP) libraries later in the book.

Semantic Web, Linked Data and Knowledge Graphs

Knowledge representation using the Semantic Web and Linked Data involves the use of web standards and technologies to represent and interlink data on the internet in a structured, machine-readable format.

We will start with a tutorial on Semantic Web data standards like RDF, RDFS, and OWL. As we saw in the setup examples in the last chapter we build Python clients using two approaches: using the Python libraries **SPARQLWrapper** and **rdflib**, and also by using general purpose Python libraries **requests** and **json**. We take a deeper dive into an example applications Knowledge Graph Creator and Knowledge Graph Navigator.

The scope of the Semantic Web is comprised of all public data sources on the Internet that follow specific standards like RDF. Knowledge Graphs may be large scale, as the graphs that drive Google's and Facebook's businesses, or they can be specific to an organization. Knowledge Graphs may be in customer-facing applications or part of internal engineering infrastructure.

The Semantic Web is a vision for the future of the World Wide Web, where data on the web is represented in a way that machines can understand, reason about, and use to perform tasks. It is built on top of the existing web and is designed to be backward-compatible. The Semantic Web relies on the use of ontologies, which are formal representations of a domain of knowledge, and the Resource Description Framework (RDF), which is a standard for expressing ontologies in a machine-readable format.

Linked Data is a set of best practices for publishing and linking data on the web using RDF. It involves the use of URIs (Uniform Resource Identifiers) to identify resources on the web and the use of links (also known as triples) to connect resources. This allows for the creation of a web of interconnected data, where each resource can be linked to other resources in a way that machines can understand and follow.

Together, the Semantic Web and Linked Data provide a framework for representing and interlinking data on the web in a structured and machine-readable format, allowing for the integration, querying, and reasoning over large amounts of information. This helps to make data on the web more accessible and useful to both humans and machines, enabling more powerful and intelligent applications and services.

As a developer you are likely familiar with the term *data lake* for enterprise-wide relational and other types of databases. Quite simply, graph databases like Fuseki and Neo4J that we previously setup are just another tool to implement *data lakes* but I prefer using the term Knowledge Graph for RDF/RDFS/OWL/SPARQL based data stores because of the ability to infer data that is not explicitly

stated as well as providing abstractions for merging different data sources in-place, that is, without the requirement to convert data to other formats or database infrastructure.

Overview and Theory

You will learn how to do the following:

- Understand RDF data formats.
- See more use cases for SPARQL queries.
- Use Python scripts to query remote SPARQL endpoints like DBPedia and WikiData as we also did in the last chapter.
- Use the SQLite relational database to cache SPARQL remote queries for both efficiency and for building systems that may have intermittent access to the Internet.
- Take a quick look at RDF, RDFS, and OWL reasoners.

The Semantic Web is intended to provide a massive linked set of data for use by software systems just as the World Wide Web provides a massive collection of linked web pages for human reading and browsing. The Semantic Web is like the web in that anyone can generate any content that they want. This freedom to publish anything works for the web because we use our ability to understand natural language to interpret what we read – and often to dismiss material that based upon our own knowledge we consider to be incorrect.

Semantic Web and linked data technologies are also useful for smaller amounts of data, an example being a Knowledge Graph containing information for a business. We will further explore Knowledge Graph use cases.

The core concept for the Semantic Web is data integration and use from different sources. As we will soon see, the tools for implementing the Semantic Web are designed for encoding data and sharing data from many different sources.

I cover the Semantic Web in this book because I believe that Semantic Web technologies are complementary to AI systems for gathering and processing data on the web. As more web pages are generated by applications (as opposed to simply showing static HTML files) it becomes easier to produce both HTML for human readers and semantic data for software agents.

RDF: The Universal Data Format

The Resource Description Framework (RDF) is used to encode information and the RDF Schema (RDFS) facilitates using data with different RDF encodings without the need to convert one set of schemas to another. Later, using OWL, we can simply declare that one predicate (or property) is the same as another; that is, one predicate is a sub-predicate of another (e.g., a property **containsCity** can be declared to be a sub-property of **containsPlace** so if something contains a city then it also contains a place), etc. The predicate part of an RDF statement often refers to a property.

RDF data was originally encoded as XML and intended for automated processing. In this chapter we will use two simple to read formats called "N-Triples" and "N3." Apache Jena can be used to convert between all RDF formats so we might as well use formats that are easier to read and understand. RDF data consists of a set of triple values:

- subject
- predicate
- object

Some of my work with Semantic Web technologies deals with processing news stories, extracting semantic information from the text, and storing it in RDF. I will use this application domain for the examples in this chapter when we implement code to automatically generate RDF for Knowledge Graphs. I deal with triples like:

- Subject: a URL (or URI) of a news article.
- Predicate (or property): a relation like "containsPerson".
- Object: a literal value like "Bill Clinton" or a URI representing Bill Clinton.

In general subjects refer to entities. In the next chapter we will use the entity recognition library we developed in an earlier chapter to create RDF from text input.

We will use either URIs or string literals as values for objects. We will always use URIs for representing subjects and predicates. In any case URIs are usually preferred to string literals. We will see an example of this preferred use but first we need to learn the N-Triple and N3 RDF formats.

I propose that the idea that RDF is more flexible than Object Modeling in programming languages, relational databases, and XML with schemas. If we can tag new attributes on the fly to existing data, how do we prevent what I might call "data chaos" as we modify existing data sources? It turns out that the solution to this problem is also the solution for encoding real semantics (or meaning) with data: we usually use unique URIs for RDF subjects, predicates, and objects, and usually with a preference for not using string literals. The definitions of predicates are tied to a namespace and later with OWL we will state the equivalence of predicates in different namespaces with the same semantic meaning. I will try to make this idea more clear with some examples and for further reference Wikipedia has a good writeup on RDF*.

Any part of a triple (subject, predicate, or object) is either a URI or a string literal. URIs encode namespaces. For example, the containsPerson predicate in the last example could be written as:

```
http://knowledgebooks.com/ontology/#containsPerson
```

The first part of this URI is considered to be the namespace for this predicate "containsPerson." When different RDF triples use this same predicate, this is some assurance to us that all users of this

*https://en.wikipedia.org/wiki/Resource_Description_Framework

predicate understand the same meaning. Furthermore, we will see later that we can use RDFS to state equivalency between this predicate (in the namespace http://knowledgebooks.com/ontology/) with predicates represented by different URIs used in other data sources. In an “artificial intelligence” sense, software that we write does not understand predicates like “containsCity”, “containsPerson”, or “isLocation” in the way that a human reader can by combining understood common meanings for the words “contains”, “city”, “is”, “person”, and “location” but for many interesting and useful types of applications that is fine as long as the predicate is used consistently. We will see that we can define abbreviation prefixes for namespaces which makes RDF and RDFS files shorter and easier to read.

There are many serialization formats for RDF (we will mostly use JSON in our Python examples):

- Turtle
- N3
- N-Triples
- NQuads
- TriG -JSON
- JSON-LD
- RDF/XML
- RDF/JSON
- TriX
- RDF Binary

A statement in N-Triple format consists of three URIs (two URIs and a string literals for the object) followed by a period to end the statement. While statements are often written one per line in a source file they can be broken across lines; it is the ending period which marks the end of a statement. The standard file extension for N-Triple format files is *.nt and the standard format for N3 format files is *.n3.

My preference is to use N-Triple format files as output from programs that I write to save data as RDF. N-Triple files don’t use any abbreviations and each RDF statement is self-contained. I often use tools like the command line commands in Jena or RDF4J to convert N-Triple files to N3 or other formats if I will be reading them or even hand editing them. Here is an example using the N3 syntax:

```
@prefix kb:  <http://knowledgebooks.com/ontology#>

<http://news.com/201234/> kb:containsCountry "China" .
```

The N3 format adds prefixes (abbreviations) to the N-Triple format. In practice it would be better to use the URI http://dbpedia.org/resource/China instead of the literal value “China.”

Here we see the use of an abbreviation prefix “kb:” for the namespace for my company Knowledge-Books.com ontologies. The first term in the RDF statement (the subject) is the URI of a news article. The second term (the predicate) is “containsCountry” in the “kb:” namespace. The last item in the

statement (the object) is a string literal "China." I would describe this RDF statement in English as, "The news article at URI http://news.com/201234 mentions the country China."

This was a very simple N3 example which we will expand to show additional features of the N3 notation. As another example, let's look at the case of this news article also mentioning the USA. Instead of adding a whole new statement like this we can combine them using N3 notation. Here we have two separate RDF statements:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234/>
  kb:containsCountry
  <http://dbpedia.org/resource/China>  .

<http://news.com/201234/>
  kb:containsCountry
  <http://dbpedia.org/resource/United_States>  .
```

We can collapse multiple RDF statements that share the same subject and optionally the same predicate:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234/>
  kb:containsCountry
    <http://dbpedia.org/resource/China> ,
    <http://dbpedia.org/resource/United_States>  .
```

The indentation and placement on separate lines is arbitrary - use whatever style you like that is readable. We can also add in additional predicates that use the same subject (I am going to use string literals here instead of URIs for objects to make the following example more concise but in practice prefer using URIs):

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234/>
        kb:containsCountry "China" ,
                           "USA" .
        kb:containsOrganization "United Nations" ;
        kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                          "Hu Jintao" , "George W. Bush" ,
                          "Pervez Musharraf" ,
                          "Vladimir Putin" ,
                          "Mahmoud Ahmadinejad" .
```

This single N3 statement represents ten individual RDF triples. Each section defining triples with the same subject and predicate have objects separated by commas and ending with a period. Please note that whatever RDF storage system you use (we will be using Jena) it makes no difference if we load RDF as XML, N-Triple, or N3 format files: internally subject, predicate, and object triples are stored in the same way and are used in the same way. RDF triples in a data store represent directed graphs that may not all be connected.

I promised you that the data in RDF data stores was easy to extend. As an example, let us assume that we have written software that is able to read online news articles and create RDF data that captures some of the semantics in the articles. If we extend our program to also recognize dates when the articles are published, we can simply reprocess articles and for each article add a triple to our RDF data store using a form like:

```
1 @prefix kb:  <http://knowledgebooks.com/ontology#> .
2
3 <http://news.com/201234/>
4   kb:datePublished
5   "2008-05-11" .
```

Note that I split one RDF statement across three lines (3-5) but this could have been on one line. The RDF statement on lines 3-5 is legal and will be handled correctly by RDF parsers. Here we just represent the date as a string. We can add a type to the object representing a specific date:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix kb:  <http://knowledgebooks.com/ontology#> .

 <http://news.com/201234/>
   kb:datePublished
   "2008-05-11"^^xsd:date .
```

Furthermore, if we do not have dates for all news articles, that is often acceptable because when constructing SPARQL queries you can match optional patterns. If for example you are looking up articles on a specific subject then some results may have a publication date attached to the results for that article and some might not. In practice RDF supports types and we would use a date type as seen in the last example, not a string. However, in designing the example programs for this chapter I decided to simplify our representation of URIs and often use string literals as simple Java strings.

Extending RDF with RDF Schema

RDF Schema (RDFS) supports the definition of classes and properties based on set inclusion. In RDFS classes and properties are orthogonal. Let's start with looking at an example using additional namespaces:

```
@prefix kb:   <http://knowledgebooks.com/ontology#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo:  <http://dbpedia.org/ontology/> .

<http://news.com/201234/>
  kb:containsCountry
  <http://dbpedia.org/resource/China>  .

<http://news.com/201234/>
  kb:containsCountry
  <http://dbpedia.org/resource/United_States>  .

<http://dbpedia.org/resource/China>
  rdfs:label "China"@en,
  rdf:type dbo:Place ,
  rdf:type dbo:Country .
```

Because the Semantic Web is intended to be processed automatically by software systems it is encoded as RDF. There is a problem that must be solved in implementing and using the Semantic Web: everyone who publishes Semantic Web data is free to create their own RDF schemas for storing data. For example, there is usually no single standard RDF schema definition for topics like news stories and stock market data. The SKOS is a namespace containing standard schemas and the most widely used standard is schema.org. Understanding the ways of integrating different data sources using different schemas helps to understand the design decisions behind the Semantic Web applications. In this chapter I often use my own schemas in the knowledgebooks.com namespace for the simple examples you see here. When you build your own production systems part of the work is searching through schema.org and SKOS to use standard name spaces and schemas when possible because this facilitates linking your data to other RDF Data on the web. The use of standard schemas helps when you link internal proprietary Knowledge Graphs used inside your organization with public open data from sources like WikiData and DBPedia.

Let's consider an example: suppose that your local Knowledge Graph referred to President Joe Biden in which case we could "mint" our own URI like:

```
https://knowledgebooks.com/person#Joe_Biden
```

In this case users of the local Knowledge Graph could not take advantage of connected data. For example, the DBPedia and WikiData URIs for Joe Biden are:

```
https://dbpedia.org/resource/Joe_Biden
http://www.wikidata.org/entity/Q6279
```

Both of these URIs can be followed by clicking on the links if you are reading a PDF copy of this book. Please "follow your nose" and see how both of these URIs resolve to human-readable web pages.

After telling you, dear reader, to always try to use public and standard URIs like the above examples for Joe Biden, I will now revert to using simple made-up URIs for the following discussion.

We will start with an example that is an extension of the example in the last section that also uses RDFS. We add a few additional RDF statements:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .
kb:containsCountry rdfs:subPropertyOf kb:containsPlace .
kb:containsState rdfs:subPropertyOf kb:containsPlace .
```

The last three lines declare that:

- The property **containsCity** is a sub-property of **containsPlace**.
- The property **containsCountry** is a sub-property of **containsPlace**.
- The property **containsState** is a sub-property of **containsPlace**.

Why is this useful? For at least two reasons:

You can query an RDF data store for all triples that use property **containsPlace** and also match triples with properties equal to **containsCity**, **containsCountry**, or **containsState**. There may not even be any triples that explicitly use the property **containsPlace**.

Consider a hypothetical case where you are using two different RDF data stores that use different properties for naming cities: **cityName** and **city**. You can define **cityName** to be a sub-property of **city** and then write all queries against the single property name **city**. This removes the necessity to convert data from different sources to use the same Schema. You can also use OWL to state property and class equivalency.

In addition to providing a vocabulary for describing properties and class membership by properties, RDFS is also used for logical inference to infer new triples, combine data from different RDF data sources, and to allow effective querying of RDF data stores. We will see examples of all of these features of RDFS when we later when using libraries to perform SPARQL queries.

The SPARQL Query Language

We briefly covered the use of SPARQL in the last chapter. SPARQL is a query language used to query RDF data stores. While SPARQL may initially look like SQL, we will see that there are some important differences like support for RDFS and OWL inferencing and graph-based instead

of relational matching operations. We will cover the basics of SPARQL in this section and then see more examples later when we learn how to embed SPARQL queries in Python applications.

We will use the N3 format RDF file test_data/news.n3 for the examples. I created this file automatically by spidering Reuters news stories on the news.yahoo.com web site and automatically extracting named entities from the text of the articles. In this chapter we use these sample RDF files.

You have already seen snippets of this file and I list the entire file here for reference, edited to fit line width: you may find the file news.n3 easier to read if you are at your computer and open the file in a text editor so you will not be limited to what fits on a book page:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .

kb:containsCountry rdfs:subPropertyOf kb:containsPlace .

kb:containsState rdfs:subPropertyOf kb:containsPlace .

<http://yahoo.com/20080616/usa_flooding_dc_16/>
        kb:containsCity "Burlington" , "Denver" ,
                        "St. Paul" ," Chicago" ,
                        "Quincy" , "CHICAGO" ,
                        "Iowa City" ;
        kb:containsRegion "U.S. Midwest" , "Midwest" ;
        kb:containsCountry "United States" , "Japan" ;
        kb:containsState "Minnesota" , "Illinois" ,
                         "Mississippi" , "Iowa" ;
        kb:containsOrganization "National Guard" ,
                         "U.S. Department of Agriculture",
                         "White House" ,
                         "Chicago Board of Trade" ,
                         "Department of Transportation" ;
        kb:containsPerson "Dena Gray-Fisher" ,
                          "Donald Miller" ,
                          "Glenn Hollander" ,
                          "Rich Feltes" ,
                          "George W. Bush" ;
        kb:containsIndustryTerm "food inflation", "food",
                                "finance ministers" ,
                                "oil" .

<http://yahoo.com/78325/ts_nm/usa_politics_dc_2/>
```

```
        kb:containsCity "Washington" , "Baghdad" ,
                        "Arlington" , "Flint" ;
        kb:containsCountry "United States" ,
                           "Afghanistan" ,
                           "Iraq" ;
        kb:containsState "Illinois" , "Virginia" ,
                         "Arizona" , "Michigan" ;
        kb:containsOrganization "White House" ,
                                "Obama administration" ,
                                "Iraqi government" ;
        kb:containsPerson "David Petraeus" ,
                          "John McCain" ,
                          "Hoshiyar Zebari" ,
                          "Barack Obama" ,
                          "George W. Bush" ,
                          "Carly Fiorina" ;
        kb:containsIndustryTerm "oil prices" .

<http://yahoo.com/10944/ts_nm/worldleaders_dc_1/>
        kb:containsCity "WASHINGTON" ;
        kb:containsCountry "United States" , "Pakistan" ,
                           "Islamic Republic of Iran" ;
        kb:containsState "Maryland" ;
        kb:containsOrganization "University of Maryland" ,
                                "United Nations" ;
        kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                          "Hu Jintao" , "George W. Bush" ,
                          "Pervez Musharraf" ,
                          "Vladimir Putin" ,
                          "Steven Kull" ,
                          "Mahmoud Ahmadinejad" .

<http://yahoo.com/10622/global_economy_dc_4/>
        kb:containsCity "Sao Paulo" , "Kuala Lumpur" ;
        kb:containsRegion "Midwest" ;
        kb:containsCountry "United States" , "Britain" ,
                           "Saudi Arabia" , "Spain" ,
                           "Italy" , India" ,
                           ""France" , "Canada" ,
                           "Russia", "Germany", "China",
                           "Japan" , "South Korea" ;
        kb:containsOrganization "Federal Reserve Bank" ,
                                "European Union" ,
```

```
                                "European Central Bank" ,
                                "European Commission" ;
        kb:containsPerson "Lee Myung-bak" , "Rajat Nag" ,
                          "Luiz Inacio Lula da Silva" ,
                          "Jeffrey Lacker" ;
        kb:containsCompany
            "Development Bank Managing" ,
            "Reuters" ,
            "Richmond Federal Reserve Bank";
        kb:containsIndustryTerm "central bank" , "food" ,
                                "energy costs" ,
                                "finance ministers" ,
                                "crude oil prices" ,
                                "oil prices" ,
                                "oil shock" ,
                                "food prices" ,
                                "Finance ministers" ,
                                "Oil prices" , "oil" .
```

Please note that in the above RDF listing I took advantage of the free form syntax of N3 and Turtle RDF formats to reformat the data to fit page width.

We will start with a simple SPARQL query for subjects (news article URLs) and objects (matching countries) with the value for the predicate equal to containsCountry. Variables in queries start with a question mark character and can have any names:

```
SELECT ?subject ?object
  WHERE {
   ?subject
   <http://knowledgebooks.com/ontology#containsCountry>
   ?object .
}
```

It is important for you to understand what is happening when we apply the last SPARQL query to our sample data. Conceptually, all the triples in the sample data are scanned, keeping the ones where the predicate part of a triple is equal to

```
http://knowledgebooks.com/ontology#containsCountry.
```

In practice RDF data stores supporting SPARQL queries index RDF data so a complete scan of the sample data is not required. This is analogous to relational databases where indices are created to avoid needing to perform complete scans of database tables.

In practice, when you are exploring a Knowledge Graph like DBPedia or WikiData (that are just very large collections of RDF triples), you might run a query and discover a useful or interesting entity URI in the triple store, then drill down to find out more about the entity. In a later example in this chapter (Knowledge Graph Navigator) we attempt to automate this exploration process using the DBPedia data as a Knowledge Graph.

We will be using the same code to access the small example of RDF statements in our sample data as we will for accessing DBPedia or WikiData.

We can make this last query easier to read and reduce the chance of misspelling errors by using a namespace prefix:

```
PREFIX kb:  <http://knowledgebooks.com/ontology#>
SELECT ?subject ?object
  WHERE {
      ?subject kb:containsCountry ?object .
  }
```

We could have filtered on any other predicate, for instance **containsPlace**. Here is another example using a match against a string literal to find all articles exactly matching the text "Maryland."

```
PREFIX kb:  <http://knowledgebooks.com/ontology#>
SELECT ?subject WHERE { ?subject kb:containsState "Maryland" . }
```

The output is:

```
http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
```

We can also match partial string literals against regular expressions:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?subject ?object
       WHERE {
         ?subject
         kb:containsOrganization
         ?object FILTER regex(?object, "University") .
       }
```

The output is:

```
http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1
  "University of Maryland"
```

We might want to return all triples matching a property of containing an organization and where the object is a string containing the substring "University." The matching statement after the FILTER check matches every triple that matches the subject in the first pattern:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT DISTINCT ?subject ?a_predicate ?an_object
 WHERE {
    ?subject kb:containsOrganization ?object .
       FILTER regex(?object,"University") .
    ?subject ?a_predicate ?an_object .
}
ORDER BY ?a_predicate ?an_object
LIMIT 10
OFFSET 5
```

When WHERE clauses contain more than one triple pattern to match, this is equivalent to a Boolean "and" operation. The DISTINCT clause removes duplicate results. The ORDER BY clause sorts the output in alphabetical order: in this case first by predicate (containsCity, containsCountry, etc.) and then by object. The LIMIT modifier limits the number of results returned and the OFFSET modifier sets the number of matching results to skip.

The output is:

```
http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
    http://knowledgebooks.com/ontology#containsOrganization
   "University of Maryland" .

http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
   http://knowledgebooks.com/ontology#containsPerson,
   "Ban Ki-moon" .

http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
   http://knowledgebooks.com/ontology#containsPerson
   "George W. Bush" .

http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
   http://knowledgebooks.com/ontology#containsPerson
   "Gordon Brown" .

http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
```

```
    http://knowledgebooks.com/ontology#containsPerson
    "Hu Jintao" .

http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
    http://knowledgebooks.com/ontology#containsPerson
    "Mahmoud Ahmadinejad" .

http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
    http://knowledgebooks.com/ontology#containsPerson
    "Pervez Musharraf" .

http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
    http://knowledgebooks.com/ontology#containsPerson
    "Steven Kull" .

http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
    http://knowledgebooks.com/ontology#containsPerson
    "Vladimir Putin" .

http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders_trust_dc_1/
    http://knowledgebooks.com/ontology#containsState
    "Maryland" .
```

We are finished with our quick tutorial on using the SELECT query form. There are three other query forms that I am not covering in this chapter:

- CONSTRUCT – returns a new RDF graph of query results.
- ASK – returns Boolean true or false indicating if a query matches any triples.
- DESCRIBE – returns a new RDF graph containing matched resources.

We will later use the OPTIONAL query form that allows complex queries to contain patterns that do not have to match.

OWL: The Web Ontology Language

We have already seen a few examples of using RDFS to define sub-properties in this chapter. The Web Ontology Language (OWL) extends the expressive power of RDFS. We now look at a few OWL examples and then look at parts of the Java unit test showing three SPARQL queries that use OWL reasoning. The following RDF data stores support at least some level of OWL reasoning:

- ProtegeOwlApis - compatible with the Protege Ontology editor.
- Pellet - DL reasoner.

- Owlim - OWL DL reasoner compatible with some versions of Sesame.
- Jena - General purpose library that is used in Apache Jena Fuseki.
- OWLAPI - a simpler API using many other libraries.
- Stardog - a commercial OWL and RDF reasoning system and datastore.
- Allegrograph - a commercial RDF+ and RDF reasoning system and datastore.

OWL is more expressive than RDFS in that it supports cardinality, richer class relationships, and Descriptive Logic (DL) reasoning. OWL treats the idea of classes very differently than object oriented programming languages like Java and Smalltalk. In OWL, instances of a class are referred to as individuals and class membership is determined by a set of properties that allow a DL reasoner to infer class membership of an individual (this is called entailment.)

We have been using the RDF file news.n3 in previous examples and we will layer new examples by adding new triples that represent RDF, RDFS, and OWL. We saw in news.n3 the definition of three triples using **rdfs:subPropertyOf** properties to create a more general **kb:containsPlace** property:

```
kb:containsCity rdfs:subPropertyOf kb:containsPlace .
kb:containsCountry rdfs:subPropertyOf kb:containsPlace .
kb:containsState rdfs:subPropertyOf kb:containsPlace .

kb:containsPlace rdf:type owl:transitiveProperty .

kbplace:UnitedStates kb:containsState kbplace:Illinois .
kbplace:Illinois kb:containsCity kbplace:Chicago .
```

We can also infer that:

```
kbplace:UnitedStates kb:containsPlace kbplace:Chicago .
```

We can also model inverse properties in OWL. For example, here we add an inverse property kb:containedIn, adding it to the example in the last listing:

```
kb:containedIn owl:inverseOf kb:containsPlace .
```

Given an RDF container that supported OWL, we can now execute SPARQL queries matching the property kb:containedIn and “match” triples in the RDF triple store that have never been asserted but are inferred by the OWL reasoner.

You should understand the concept of class membership not by explicitly stating that an object (or individual) is a member of a class, but rather because an individual has properties that can be used to infer class membership.

The World Wide Web Consortium has defined three versions of the OWL language that are in increasing order of complexity: OWL Lite, OWL DL, and OWL Full. OWL DL (supports Description Logic) is the most widely used (and recommended) version of OWL. OWL Full is not computationally decidable since it supports full logic, multiple class inheritance, and other things that probably make it computationally intractable for all but smaller problems.

A Hybrid Deep Learning and RDF/SPARQL Application for Question Answering

We will skip ahead a little and use two deep learning models (spaCy NLP and Transformer) with SPARQL queries to answer natural language questions. I wrote this example in January 2001 and it generated quite a lot of interest on social media and I later noticed several projects that picked up my basic idea of:

- Using spaCy to identify proper nouns in text (e.g., human names, locations, corporations, etc.).
- Use SPARQL queries to collect text describing all identified proper nouns.
- Use a question answering Transformer model with two inputs: the user's question and the text collected by the SPARQL queries.

Note: this example is now somewhat obsolete since GPT-3 (that we use later in the book) and ChatGPT models can answer questions directly. Still, the example we use here is very simple and "hackable" and I hope you enjoy it.

You can access this example on Google Colab Colab DBPedia Sparql Question Answering Demo*.

*https://colab.research.google.com/drive/1FX-0eizj2vayXsqfSB2ONuJYG8BaYpGO?usp=sharing

DBPedia Question Answering System Copyright 2020-2021 Mark Watson. All rights reserved. License: Apache 2 https://markwatson.com

```
%%capture
!pip install transformers
!pip install SPARQLWrapper
!pip freeze

from transformers import pipeline

qa = pipeline(
    "question-answering",
    #model="NeuML/bert-small-cord19qa",
    model="NeuML/bert-small-cord19-squad2",
    tokenizer="NeuML/bert-small-cord19qa"
)

!pip install import spacy
!python -m spacy download en_core_web_sm
```

```
import spacy

nlp_model = spacy.load('en_core_web_sm')
```

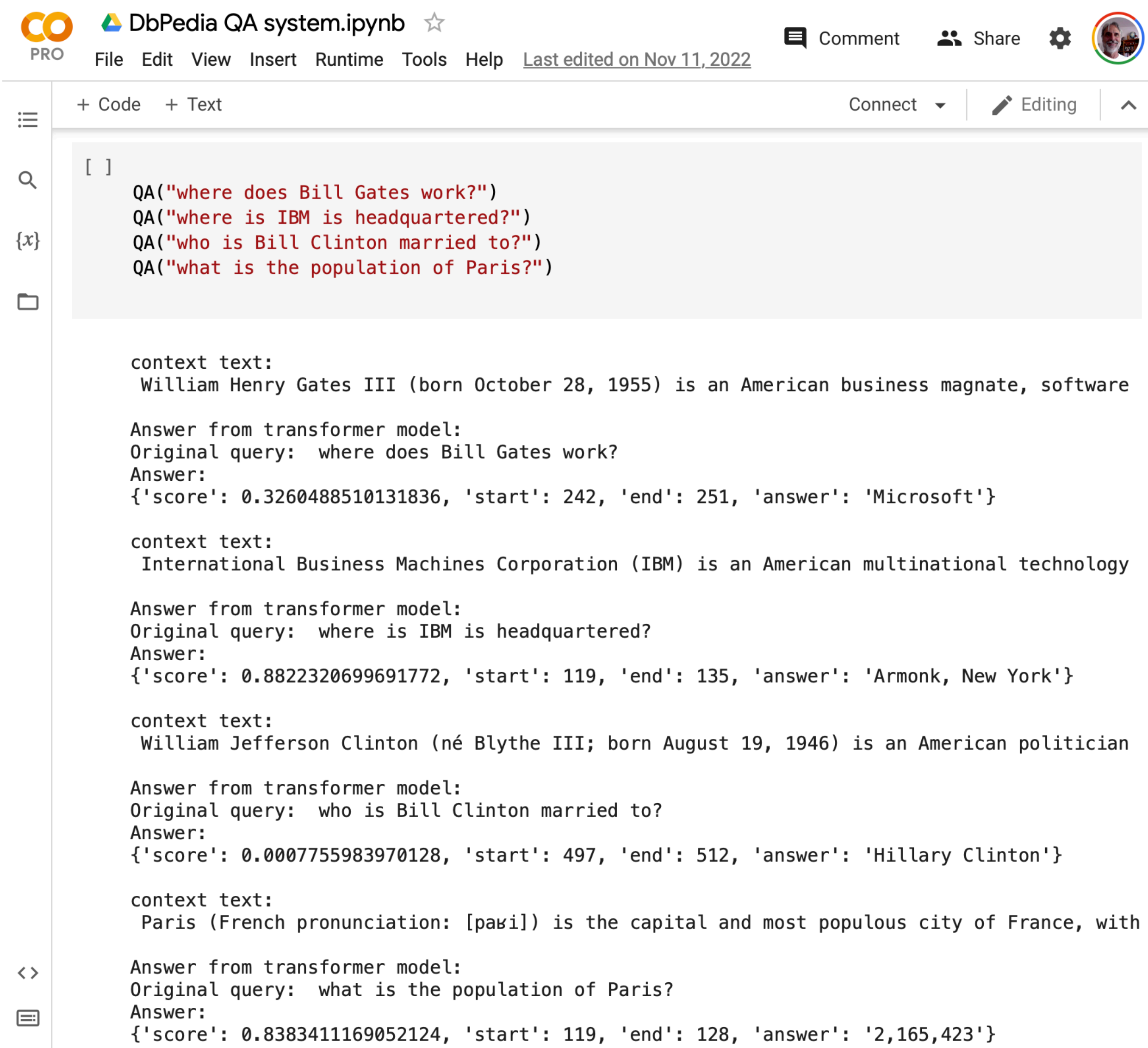
```
from SPARQLWrapper import SPARQLWrapper, JSON

sparql = SPARQLWrapper("http://dbpedia.org/sparql")

def query(query):
  sparql.setQuery(query)
  sparql.setReturnFormat(JSON)
  return sparql.query().convert()['results']['bindings']
```

Here we install a pre-trained deep learning model that can answer questions, given text that contains the answers to the questions. We then import the spaCy NLP library. Finally, I defined a simple utility function for making SPARQL queries.

```python
def entities_in_text(s):
    doc = nlp_model(s)
    ret = {}
    for [ename, etype] in [[entity.text, entity.label_] for entity in doc.ents]:
        if etype in ret:
            ret[etype] = ret[etype] + [ename]
        else:
            ret[etype] = [ename]
    return ret

def dbpedia_get_entities_by_name(name, dbpedia_type):
  sparql = "select distinct ?s ?comment where {{ ?s <http://www.w3.org/2000/01/rdf-schema#label>
  #print(sparql)
  results = query(sparql)
  return(results)

entity_type_to_type_uri = {'PERSON': '<http://dbpedia.org/ontology/Person>',
    'GPE': '<http://dbpedia.org/ontology/Place>', 'ORG':
    '<http://dbpedia.org/ontology/Organisation>'}
```

```python
def QA(query_text):
  entities = entities_in_text(query_text)

  def helper(entity_type):
    ret = ""
    if entity_type in entities:
      for hname in entities[entity_type]:
        results = dbpedia_get_entities_by_name(hname, entity_type_to_type_uri[entity_type])
        for result in results:
          ret += ret + result['comment']['value'] + " . "
    return ret

  context_text = helper('PERSON') + helper('ORG') + helper('GPE')
  print("\ncontext text:\n", context_text, "\n")

  print("Answer from transformer model:")
  print("Original query: ", query_text)
  print("Answer:")

  answer = qa({
              "question": query_text,
              "context": context_text
             })
  print(answer)
```

There are two functions defined here. **entities_in_text** uses spaCy to find entities in text and return both the entities and the entity types. The top level function **QA** takes a question as an input, finds the entities, collects text from DBPedia about those entities, and then uses the pre-trained question answering deep learning model to answer the input question.



Here we used the top level function **QA** with a few sample questions.

Knowledge Graph Creator: Convert Text Files to RDF Data Input Data for Fuseki

I published my **kgcreator** command line Python app to PyPy: https://pypi.org/project/kgcreator/. The GitHub repository is https://github.com/mark-watson/kgcreator.

You can install my **kgcreator** library with:

```
pip install kgcreator
pip install spacy
python -m spacy download en_core_web_sm
```

If you clone the repository for **kgcreator** then you can use the test input files in the directory **test_-data** to run the program, first to see the programs command line options and then to run it on the test input files:

```
kgcreator --help
kgcreator --inputdir=test_data --outputfile=out.rdf  --outputfileneo4j=out.cypher
```

Both RDF data and Cypher data for Neo4J are written to local output files. The main library file **kgcreator.py** is short enough to list and discuss. In lines 6-12 we are loading a spaCy English language model. If the models not been downloaded it the load operation throws an exception and we download the model.

The function **find_entities_in_text** (lines 14-19) uses the spaCy library to find entities like people, organizations, etc. The function **data2Rdf** (lines 21-27) requires three arguments: meta data for the file path, a list of value/abbreviation sublists, and an output file stream to write RDF data to. The function **data2Cypher** (lines 29-45) takes the same arguments but instead writes the data in Cypher format for Neo4J:

```
1  from os import scandir
2  from os.path import splitext, exists
3  from pprint import pprint
4  import spacy
5
6  try:
7    nlp_model = spacy.load('en_core_web_sm')
8  except:
9    print("Loading spaCy model file...")
10   from os import system
11   system("python -m spacy download en_core_web_sm")
12   nlp_model = spacy.load('en_core_web_sm')
13
14 def find_entities_in_text(some_text):
15
16     def clean(s):
17         return s.replace('\n', ' ').strip()
18     doc = nlp_model(some_text)
19     return map(list, [[clean(entity.text), entity.label_] for entity in doc.ents])
20
21 def data2Rdf(meta_data, entities, fout):
```

```
    for [value, abbreviation] in entities:
        a_literal = '"' + value + '"'
        if value in v2umap:
            a_literal = v2umap[value]
        fout.write('<' + meta_data + '>\t' + e2umap[abbreviation] + '\t' +
            a_literal + ' .\n') if abbreviation in e2umap else None

def data2Cypher(meta_data, entities, fout):
    SUMMARY = "To Be Done"
    for [name, atype] in entities:
        if atype in e2umap:
            fout.write('CREATE (' + name.replace(" ", '_') +
                       ':CategoryType {name:' + e2umap[atype] + '})\n')
    # start by creating a node for source URI:
    meta_print_name = meta_data[meta_data.index('//') + 2:]
    fout.write('CREATE (' + meta_print_name + ':News {name:"' + meta_print_name +
               '", uri: "' + meta_data + '", summary: "' + SUMMARY + '"})\n')

    for [name, atype] in entities:
        fout.write('CREATE (' + name.replace(" ", '_') + ')-[:Category]->(' + e2umap\
[atype] + '))\n')
        fout.write('CREATE (' + meta_print_name + ')-[:' + e2umap[atype] + ')->(' + \
name.replace(" ", '_') + ')\n')

e2umap = {'ORG': '<https://schema.org/Organization>',
          'LOC': '<https://schema.org/location>',
          'GPE': '<https://schema.org/location>',
          'NORP': '<https://schema.org/nationality>',
          'PRODUCT': '<https://schema.org/Product>',
          'PERSON': '<https://schema.org/Person>'}
v2umap = {'IBM': '<http://dbpedia.org/page/IBM>',
          'The Wall Street Journal': '<http://dbpedia.org/page/The_Wall_Street_Journ\
al>',
          'Banco Espirito': '<http://dbpedia.org/page/Banco_Esp%C3%ADrito_Santo>',
          'Australian Broadcasting Corporation': '<http://dbpedia.org/page/Australia\
n_Broadcasting_Corporation>',
          'Australian Writers Guild': '<http://dbpedia.org/page/Australian_Broadcast\
ing_Corporation>',
          'Microsoft': '<http://dbpedia.org/page/Microsoft>'}

def process_directory(directory_name, output_rdf, output_neo4j):
    with open(output_rdf, 'w') as frdf:
        with open(output_neo4j, 'w') as fneo4j:
```

```
65             with scandir(directory_name) as entries:
66                 for entry in entries:
67                     [_, file_extension] = splitext(entry.name)
68                     if file_extension == '.txt':
69                         check_file_name = entry.path[0:-4:None] + '.meta'
70                         if exists(check_file_name):
71                             process_file(entry.path, check_file_name, frdf, fneo4j)
72                         else:
73                             print('Warning: no .meta file for', entry.path, 'in dire\
74 ctory', directory_name)
75 
76 def process_file(txt_path, meta_path, frdf, fneo4j):
77 
78     print(f"** process_file txt_path={txt_path} meta_path={meta_path}")
79     def read_data(text_path, meta_path):
80         with open(text_path) as f:
81             t1 = f.read()
82         with open(meta_path) as f:
83             t2 = f.read()
84         return [t1, t2]
85 
86     def modify_entity_names(ename):
87         return ename.replace('the ', '')
88 
89     [txt, meta] = read_data(txt_path, meta_path)
90     entities = find_entities_in_text(txt)
91     entities = [[modify_entity_names(e), t] for [e, t] in entities if t in
92         ['NORP', 'ORG', 'PRODUCT', 'GPE', 'PERSON', 'LOC']]
93     data2Rdf(meta, entities, frdf)
94     data2Cypher(meta, entities, fneo4j)
95 
96 # process_directory('../test_data', 'out.rdf', 'out.cypher')
```

The dictionary object **e2umap** defined in lines 47-52 maps a spaCy entity type name to a URI, for example '**PERSON**': '**https://schema.org/Person**'. The dictionary object **v2umap** defined in lines 53-61 maps names to common organizations to URIs. Note that this map can be extended for additional entity names. If **v2umap** does not contain an organization name then the organization will be output as a string literal and not a URI.

The top-level function **process_directory** (lines 63-75) reads all text files in an input directory and calls the helper functions we have already discussed to create output RDF and Cypher files using the helper function **process_file**.

Old Technology: The OpenCyc Knowledge Base (Optional Material)

Here we use the free version of the OpenCyc Knowledge Base. OpenCyc is no longer supported by the Cyc corporation (they still sell commercial versions). I still find this knowledge base useful and here we use a version that has been converted to RDF data.

After loading the OpenCyc data as RDF into Apache Fuseki and exploring the data we will use the SPARQL SERVICE operator to combine data from our local server with the public DBPedia Knowledge Graph.

Adam Sanchez has a GitHub repository that contains the OpenCyc OWL/RDF files*. While I tried to make this section self-contained and interesting to just read, if you want to experiment with the latest OpenCyc 4.0 OWL/RDF dataset then download this file† and follow along on your laptop.

I base the material in this section on an old blog article I wrote in 2014 Using OpenCyc RDF/OWL data in StarDog ‡ that showed how to import the OpenCyc OWL/RDF files into the commercial RDF datastore Stardog. Here I do much the same thing using Apache Jena/Fuseki but we will dive in deeper than the original article.

If you have downloaded the latest OpenCyc OWL file then it can be loaded by:

```
./fuseki-server --file /Users/markw/OpenCyc_owl_rdf/opencyc-latest.owl /opencyc
```

As I did in my blog article, we start by using a SPARQL query that contains "Clinton" as the object in a triple and we get 207 triples from this query:

```
SELECT ?s ?p ?o WHERE { ?s ?p ?o FILTER(REGEX(?o, "Clinton")) } LIMIT 500
```

This triple identifies the OpenCyc subject for Hilliary Clinton:

```
<http://sw.opencyc.org/concept/Mx4rvV7SqpwpEbGdrcN5Y29ycA>
    <http://sw.cyc.com/CycAnnotations_v1#label>
    "HillaryClinton"@en .
```

*https://github.com/asanchez75/opencyc
†https://www.amazon.com/clouddrive/share/urtlDhQbmeMz24TUNED3KiyzrqOlMYZ5gdLpTTSdcFR
‡https://mark-watson.blogspot.com/2014/07/using-opencyc-rdfowl-data-in-stardog.html

```
SELECT ?p ?o
WHERE {
  <http://sw.opencyc.org/concept/Mx4rvV7SqpwpEbGdrcN5Y29ycA>
  ?s ?p
} LIMIT 500
```

Of particular use is the matched result:

```
?p  ?o
<http://www.w3.org/2002/07/owl#sameAs>
  <http://dbpedia.org/resource/Hillary_Rodham_Clinton>
```

This lets us combine data from OpenCyc with DBPedia (limit to just 2500 results). This is not a particularly good example since we are in no way tying together data from OpenCyc to DBPedia (we will combine the results later), rather we are just doing two separate queries:

```
SELECT *
WHERE {
  <http://sw.opencyc.org/concept/Mx4rvV7SqpwpEbGdrcN5Y29ycA> ?p ?o .

  SERVICE <http://dbpedia.org/sparql?timeout=30000> {
    <http://dbpedia.org/resource/Hillary_Rodham_Clinton>
      ?p_dbpedia_1
      ?o_dbpedia_1 .
    ?s_dbpedia_2
      ?p_dbpedia_2
      <http://dbpedia.org/resource/Hillary_Rodham_Clinton> .
  }
} limit 2500
```

Two results chosen that only used English language (DBPedia has triples containing text in many human languages):

```
?p  ?o  ?p_dbpedia_1    ?o_dbpedia_1    ?s_dbpedia_2    ?p_dbpedia_2

<http://www.w3.org/2002/07/owl#sameAs>
  <http://dbpedia.org/resource/Hillary_Rodham_Clinton>
  <http://www.w3.org/2000/01/rdf-schema#label>
  "Hillary Clinton"@en
  <http://dbpedia.org/resource/American_Academy_of_Arts_and_Sciences_members>
  <http://dbpedia.org/ontology/wikiPageWikiLink>

```

```
<http://www.w3.org/2002/07/owl#sameAs>
  <http://dbpedia.org/resource/Hillary_Rodham_Clinton>
  <http://www.w3.org/2000/01/rdf-schema#label>
  "Hillary Rodham Clinton"@en
  <http://dbpedia.org/resource/Lincoln_Bedroom_for_contributors_controversy>
  <http://dbpedia.org/ontology/wikiPageWikiLink>
```

If we don’t link data from two RDF services then we are obviously better off doing two separate queries and combining the results in our application.

Before linking data for OpenCyc and DBPedia, let’s look at a Python SPARQL query example that just access OpenCyc RDF data on a local Fuseki server:

```python
import rdflib
from SPARQLWrapper import SPARQLWrapper, JSON
from pprint import pprint

queryString = """
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT *
WHERE {
    <http://sw.opencyc.org/concept/Mx4rvV7SqpwpEbGdrcN5Y29ycA>
      rdfs:label ?label
      FILTER (lang(?label) = 'en') .
    <http://sw.opencyc.org/concept/Mx4rvV7SqpwpEbGdrcN5Y29ycA>
      <http://www.w3.org/2002/07/owl#sameAs> ?dbpedia_uri
      filter(strstarts(str(?dbpedia_uri), "http://dbpedia.org/resource")) .
}
LIMIT 5
"""

sparql = SPARQLWrapper("http://localhost:3030/opencyc")
sparql.setQuery(queryString)
sparql.setReturnFormat(JSON)
sparql.setMethod('POST')
ret = sparql.queryAndConvert()
for r in ret["results"]["bindings"]:
    pprint(r)
```

The output is:

```
$ python opencyc_example_1.py
{'dbpedia_uri': {'type': 'uri',
                 'value': 'http://dbpedia.org/resource/Hillary_Rodham_Clinton'},
 'label': {'type': 'literal', 'value': 'Hillary Clinton', 'xml:lang': 'en'}}
```

Let's now link local SPARQL results from OpenCyc with information from DBPedia. We replace the **queryString** variable value in the last code listing with:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT *
WHERE {
    <http://sw.opencyc.org/concept/Mx4rvV7SqpwpEbGdrcN5Y29ycA>  rdfs:label ?label
    FILTER (lang(?label) = 'en') .
    <http://sw.opencyc.org/concept/Mx4rvV7SqpwpEbGdrcN5Y29ycA>
      <http://www.w3.org/2002/07/owl#sameAs> ?dbpedia_uri
      filter(strstarts(str(?dbpedia_uri), "http://dbpedia.org/resource")) .
  SERVICE <http://dbpedia.org/sparql?timeout=30000> {
      ?dbpedia_uri  ?dbpedia_property ?dbpedia_object  .
  }
}
LIMIT 4
```

The output is:

```
$ python opencyc_example_2.py
{'dbpedia_object': {'type': 'literal',
                    'value': 'Hillary Rodham Clinton',
                    'xml:lang': 'en'},
 'dbpedia_property': {'type': 'uri',
                      'value': 'http://www.w3.org/2000/01/rdf-schema#label'},
 'dbpedia_uri': {'type': 'uri',
                 'value': 'http://dbpedia.org/resource/Hillary_Rodham_Clinton'},
 'label': {'type': 'literal', 'value': 'Hillary Clinton', 'xml:lang': 'en'}}

{'dbpedia_object': {'type': 'literal',
                    'value': 'Hillary Clinton',
                    'xml:lang': 'ca'},
 'dbpedia_property': {'type': 'uri',
                      'value': 'http://www.w3.org/2000/01/rdf-schema#label'},
 'dbpedia_uri': {'type': 'uri',
                 'value': 'http://dbpedia.org/resource/Hillary_Rodham_Clinton'},
```

```
 'label': {'type': 'literal', 'value': 'Hillary Clinton', 'xml:lang': 'en'}}
{'dbpedia_object': {'type': 'literal',
                    'value': 'Hillary Clintonova',
                    'xml:lang': 'cs'},
 'dbpedia_property': {'type': 'uri',
                      'value': 'http://www.w3.org/2000/01/rdf-schema#label'},
 'dbpedia_uri': {'type': 'uri',
                 'value': 'http://dbpedia.org/resource/Hillary_Rodham_Clinton'},
 'label': {'type': 'literal', 'value': 'Hillary Clinton', 'xml:lang': 'en'}}
```


We can use the SPARQL OPTIONAL operator to match data patterns that may or may not exist. OPTIONAL is a binary operator that combines two graph patterns:

```
SELECT *
WHERE {
    <http://sw.opencyc.org/concept/Mx4rvV7SqpwpEbGdrcN5Y29ycA>  rdfs:label ?label
    FILTER (lang(?label) = 'en') .
    <http://sw.opencyc.org/concept/Mx4rvV7SqpwpEbGdrcN5Y29ycA>
      <http://www.w3.org/2002/07/owl#sameAs> ?dbpedia_uri
      filter(strstarts(str(?dbpedia_uri), "http://dbpedia.org/resource")) .
  SERVICE <http://dbpedia.org/sparql?timeout=15000> {
    ?dbpedia_uri  rdfs:label ?dbpedia_label
      FILTER (lang(?dbpedia_label) = 'en') .
    OPTIONAL {
       ?dbpedia_uri  rdfs:comment ?dbpedia_comment
       FILTER (lang(?dbpedia_comment) = 'en')
    } .
  }
}
```

When I need to collect text on an entity I often look for comment data on DBPedia (as we did in the previous **kgcreator** example). In this case, there were no English language comments so no comment results are in the returned JSON:


~~~
 $ python opencyc_example_3.py
{'dbpedia_label': {'type': 'literal',
                   'value': 'Hillary Rodham Clinton',
                   'xml:lang': 'en'},
 'dbpedia_uri': {'type': 'uri',
                 'value': 'http://dbpedia.org/resource/Hillary_Rodham_Clinton'},
 'label': {'type': 'literal', 'value': 'Hillary Clinton', 'xml:lang': 'en'}}```
~~~

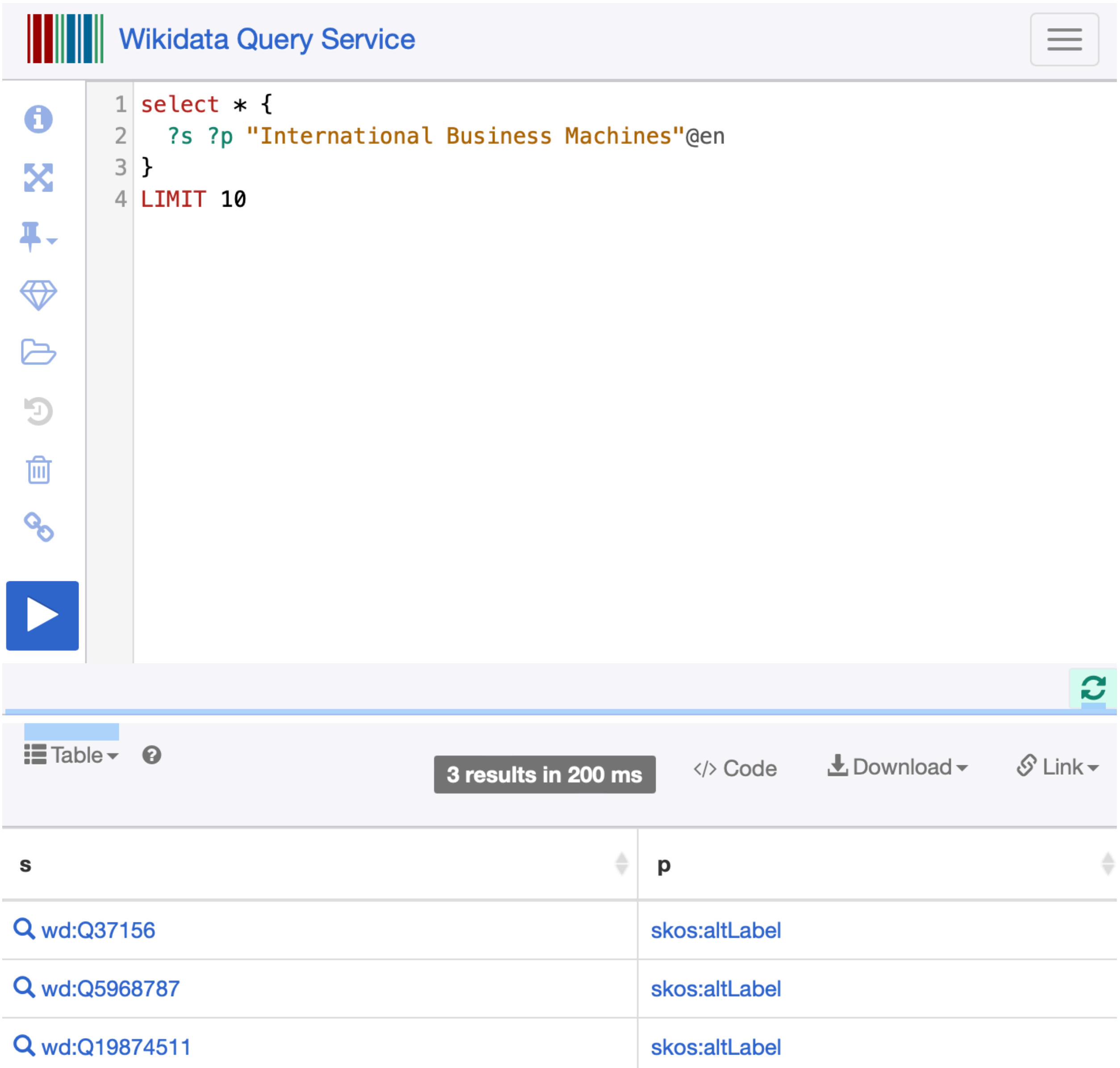
If we didn't use the OPTIONAL operator then we would not have retrieved data for the first pattern in the WHERE clause.

We now leave our discussion of using the no longer updated OpenCyc data and look at Python code in the next section that uses the Wikidata SPARQL server rather than DBPedia.

Examples Using Wikidata Instead of DBPedia

Wikidata uses abstract URIs instead of human readable URIs that DBPedia uses. Because of Wikidata's abstract URIs I usually use DBPedia when experimenting with new ideas. That said, there is more data in Wikidata. The examples in this section will get you started if you want to experiment with Wikidata.

As with DBPedia, start with Wikidata's public SPARQL endpoint https://query.wikidata.org. I want to walk you through resolving abstract URIs to something human readable by starting with a SPARQL query:



In the SPARQL results there are three matching subjects:

- wd:Q37156* that has the URI value of https://www.wikidata.org/wiki/Q37156. Click on this link. This is the entity we want but also try clicking on these links:
- wd:Q5968787†
- wd:Q19874511‡

*https://www.wikidata.org/wiki/Q37156
†https://www.wikidata.org/wiki/Q5968787
‡https://www.wikidata.org/wiki/Q19874511

Here is a simple Python program that uses Wikidata. In this example we make multiple SPARQL queries: first to find WikiData URIs for IBM, and then to find all **label** property values for each of these URIs:

```
## Test client for Wikidata SPARQL endpoint

from SPARQLWrapper import SPARQLWrapper, JSON
from pprint import pprint

queryString = """
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT *
WHERE {
  ?subject skos:altLabel "International Business Machines"@en .
}
LIMIT 4
"""

uris = []

sparql = SPARQLWrapper("https://query.wikidata.org/sparql")
sparql.setQuery(queryString)
sparql.setReturnFormat(JSON)
ret = sparql.query().convert()
for r in ret["results"]["bindings"]:
  pprint(r)
  if 'subject' in r:
    if 'value' in r['subject']:
      uri = r['subject']['value']
      print(uri)
      uris.append(uri)

queryString2 = """
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT *
WHERE {
  <A_URI> wdt:P31 ?entity_label . # wdt:P31 is instanceOf
  ?entity_label skos:altLabel ?entity_human_readable_label
    FILTER (lang(?entity_human_readable_label) = 'en') .
}
LIMIT 5
```

```
"""

def wd_helper(an_ibm_uri):
    print(f"\n *** {an_ibm_uri} ***\n")
    query = queryString2.replace("A_URI", an_ibm_uri)
    #print(query)
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    ret = sparql.query().convert()['results']['bindings']
    for r in ret:
      print(r['entity_human_readable_label']['value'])

for uri in uris:
  wd_helper(uri)
```

The output is:

```
$ python wikidata1.py
{'subject': {'type': 'uri', 'value': 'http://www.wikidata.org/entity/Q37156'}}
http://www.wikidata.org/entity/Q37156
{'subject': {'type': 'uri', 'value': 'http://www.wikidata.org/entity/Q5968787'}}
http://www.wikidata.org/entity/Q5968787
{'subject': {'type': 'uri',
             'value': 'http://www.wikidata.org/entity/Q19874511'}}
http://www.wikidata.org/entity/Q19874511

 *** http://www.wikidata.org/entity/Q37156 ***

device
hw
hardware
computer component
computer accessory

 *** http://www.wikidata.org/entity/Q5968787 ***

edifice
buildings

 *** http://www.wikidata.org/entity/Q19874511 ***

lab
research laboratory
```

```
research facility
research lab
laboratories
```

I would like you to have a few takeaways from this material:

- When using public Knowledge Graphs like DBPedia and Wikipedia, you want to start by using the public SPARQL endpoints to explore the data to understand what might be useful for your project.
- Write low-level libraries to make SPARQL queries and filter and transform the JSON query results data to a form that you can easily use.
- Given a foundation of data access and transformation tools, then write your application.

In the next section we look at a tool I wrote for exploring Knowledge Graphs.

Knowledge Graph Navigator: Use English to Explore DBPedia

When I need to use DBPedia data in applications, before I start writing any code I explore possibly useful information using:

- https://dbpedia.org/sparql DBPedia SPARQL endpoint.
- https://dbpedia.org/fct/ OPEN LINK Software text search and entity search.

The following example is my effort to create a tool that quickly identifies entities like people, places, and organization and the relations between these discovered entities.

I published the **kgn** command line Python app to PyPy: https://pypi.org/project/kgn/. The GitHub repository is https://github.com/mark-watson/kgn.

We will look at a few snippets of the code. Here is a roadmap by source file in alphabetical order:

- cache.py caches SPARQL query results in an SQLite database.
- cli.py is the top level command line tool.
- colorize.py colorizes generated SPARQL queries to make them more readable.
- kgn.py is the main logic for the Knowledge Graph Navigator.
- kgnutils.py contains a function for resolving text entity names into DBPedia URIs.
- relationships.py takes a list of N entity URIs and performs an exhaustive search to find relationships between pairs of entities. This code runs O(N^2) so it is best to not input more than 5 or 6 text entity names.
- sparql.py is a collection of reusable SPARQL utilities.

- textui.py contains helper functions for the text-based user interface.

This is a fairly long example but if you followed the previous Python + SPARQL query examples, then it should be fairly clear how this works. When we identify entities in input text we generate SPARQL queries to match the literal entity names. If this is possible, then we have the DBPedia URIs for entities in the input text and it is straightforward to get comment text for entities and search for properties (relationships) that link any two entity URIs using a SPARQL matching pattern like:

```
  <entity_1_URI> ?p <entity_2_URI> .
```

Listing of the main application logic in **kgn.py**:

```
from pprint import pprint
from .kgnutils import dbpedia_get_entities_by_name
from .textui import select_entities, get_query
from .relationships import entity_results_to_relationship_links
import spacy

try:
  nlp_model = spacy.load('en_core_web_sm')
except:
  print("Loading spaCy model file...")
  from os import system
  system("python -m spacy download en_core_web_sm")
  nlp_model = spacy.load('en_core_web_sm')


def entities_in_text(s):
    " use spaCY to find entity names in text "
    doc = nlp_model(s)
    ret = {}
    for [ename, etype] in [[entity.text, entity.label_] for entity in doc.ents]:
        if etype in ret:
            ret[etype] = ret[etype] + [ename]
        else:
            ret[etype] = [ename]
    return ret

entity_type_to_type_uri = {'PERSON': '<http://dbpedia.org/ontology/Person>',
    'GPE': '<http://dbpedia.org/ontology/Place>', 'ORG':
    '<http://dbpedia.org/ontology/Organisation>'}
short_comment_to_uri = {}
```

```python

def shorten_comment(comment, uri):
    sc = comment[0:70:None] + '...'
    short_comment_to_uri[sc] = uri
    return sc

query = ''

def kgn():
    print("Knowledge Graph Navigator (note: only runs in a terminal)")
    while True:
        query = get_query()
        if query == 'quit' or query == 'q':
            break
        elist = entities_in_text(query)
        people_found_on_dbpedia = []
        places_found_on_dbpedia = []
        organizations_found_on_dbpedia = []
        global short_comment_to_uri
        short_comment_to_uri = {}
        for key in elist:
            type_uri = entity_type_to_type_uri[key]
            for name in elist[key]:
                dbp = dbpedia_get_entities_by_name(name, type_uri)
                for d in dbp:
                    short_comment = shorten_comment(d[1][1], d[0][1])
                    people_found_on_dbpedia.extend([name + ' || ' +
                        short_comment]) if key == 'PERSON' else None
                    places_found_on_dbpedia.extend([name + ' || ' +
                        short_comment]) if key == 'GPE' else None
                    organizations_found_on_dbpedia.extend([name + ' || ' +
                        short_comment]) if key == 'ORG' else None
        user_selected_entities = select_entities(people_found_on_dbpedia,
            places_found_on_dbpedia, organizations_found_on_dbpedia)
        uri_list = []
        for entity in user_selected_entities['entities']:
            short_comment = entity[4 + entity.index(' || '):None:None]
            uri_list.extend([short_comment_to_uri[short_comment]])
        print("\n\nEntity data:")
        pprint(user_selected_entities)
        print("\n\n")
        relation_data = (
            entity_results_to_relationship_links(uri_list))
```

```
74         print('\n\nDiscovered relationship links:\n')
75         for relationship in relation_data:
76             print(relationship[0] + ' --> ' + relationship[2][1] +
77                   ' --> ' + relationship[1])
```

This command line tool does not run very well in a shell in IDEs like PyCharm so pay attention to the printed prompt in line 40 and run **kgn** in a terminal window that properly renders unicode characters and colored/styled text.

A listing of **kgnutils.py** that uses a SPARQL query to resolve entity names to DBPedia URIs:

```
from .sparql import dbpedia_sparql
from .colorize import colorize_sparql

def dbpedia_get_entities_by_name(name, dbpedia_type):
    sparql = (
        'select distinct ?s ?comment {{ ?s ?p "{}"@en . ?s <http://www.w3.org/2000/0\
1/rdf-schema#comment>  ?comment  . FILTER  (lang(?comment) = \'en\') . ?s <http://ww\
w.w3.org/1999/02/22-rdf-syntax-ns#type> {} . }} limit 15'
        .format(name, dbpedia_type))
    print('Generated SPARQL to get DBPedia entity URIs from a name:')
    print(colorize_sparql(sparql))
    return dbpedia_sparql(sparql)
```

The listing of **relationships.py** that finds RDF properties that link any two entity URIs:

```
from .sparql import dbpedia_sparql
from .colorize import colorize_sparql

def flatten(a_list):
    return [item for items in a_list for item in items]

def dbpedia_get_relationships(s_uri, o_uri):
    query = (
        "SELECT DISTINCT ?p {{  {} ?p {} . FILTER (!regex(str(?p), 'wikiPage', 'i'))\
 }} LIMIT 5"
        .format(s_uri, o_uri))
    results = dbpedia_sparql(query)
    print('Generated SPARQL to get relationships between two entities:')
    print(colorize_sparql(query))
    return [r for r in flatten(results) if not r == 'p']


```

```python
def entity_results_to_relationship_links(uris):
    uris = [('<' + uri + '>') for uri in uris]
    relationship_statements = []
    for e1 in uris:
        for e2 in uris:
            if not e1 == e2:
                l1 = dbpedia_get_relationships(e1, e2)
                l2 = dbpedia_get_relationships(e2, e1)
                for x in l1:
                    relationship_statements.extend([[e1, e2, x]]) if not [e1,
                        e2, x] in relationship_statements else None
                for x in l2:
                    relationship_statements.extend([[e1, e2, x]]) if not [e1,
                        e2, x] in relationship_statements else None
    return relationship_statements
```

A listing of **sparql.py** that is a utility to query either the DBPedia or Wikidata SPARQL server endpoints:

```python
import requests
from .cache import fetch_result_dbpedia, save_query_results_dbpedia
wikidata_endpoint = 'https://query.wikidata.org/bigdata/namespace/wdq/sparql'
dbpedia_endpoint = 'https://dbpedia.org/sparql'


def do_query_helper(endpoint, query):
    cached_results = fetch_result_dbpedia(query)
    if len(cached_results) > 0:
        print('Using cached query results')
        return cached_results # eval(cached_results)

    params = {'query': query, 'format': 'json'}
    response = requests.get(endpoint, params=params)
    json_data = response.json()
    vars = json_data['head']['vars']
    results = json_data['results']
    if 'bindings' in results:
        bindings = results['bindings']
        qr = [[[var, binding[var]['value']] for var in vars] for binding in bindings]
        save_query_results_dbpedia(query, qr)
        return qr
    return []

```

```python

def wikidata_sparql(query):
    return do_query_helper(wikidata_endpoint, query)


def dbpedia_sparql(query):
    return do_query_helper(dbpedia_endpoint, query)
```

This example Python code for performing SPARQL queries differs from the previous examples that all used the **SPARQLWrapper** library. Here I used the Python **requests** library.

This program produces a lot of printed output so I refer you to the GitHub repository for my command line tool **kgn** https://github.com/mark-watson/kgn for example usage in the top level README file.

Wrap Up for Semantic Web, Linked Data and Knowledge Graphs

I hope that you both enjoyed this chapter and that it has some practical use for you either in personal or professional projects. I favored the use of the open source Apache Jena/Fuseki platform. It is not open source but the free to use version of Ontotext GraphDB[*] has interesting graph visualization tools that you might want to experiment with. I also sometime use the commercial products Franz AllegorGraph[†] and Stardog[‡].

The Python examples in this chapter are simple examples to get you started. In real projects I build a library of low-level utilities to manipulate the JSON data returned from SPARQL endpoints. As an example, I almost always write filters for removing data that is text but not in English. This filtering is especially important for Wikidata that has most data replicated for most human written languages.

[*] https://www.ontotext.com/products/graphdb/
[†] https://allegrograph.com
[‡] https://www.stardog.com/platform

Part III - Deep Learning

In the next two chapters we explore some basic theory underlying deep learning and then look at practical examples building models from spreadsheet data, performing natural language processing (NLP) tasks, and have fun with models to generate images from text.

When you have finished reading this section and want to learn more about specific deep learning architectures I recommend using this up to date list of short descriptive papers https://github.com/dair-ai/ML-Papers-Explained.

The Basics of Deep Learning

Deep learning is a subfield of machine learning that is concerned with the design and implementation of artificial neural networks (ANNs) with multiple layers, also known as deep neural networks (DNNs). These networks are inspired by the structure and function of the human brain, and are designed to learn from large amounts of data such as images, text, and audio.

A neural network consists of layers of interconnected nodes, or neurons, which are organized into an input layer, one or more hidden layers, and an output layer. Each neuron receives input from the neurons in the previous layer, performs a computation, and passes the result to the neurons in the next layer. The computation typically involves a dot product of the input with a set of weights and an activation function, which is a non-linear function applied to the result. The weights are the parameters of the network that are learned during training.

The basic building block of a deep neural network is an artificial neuron, also known as perceptron, which is a simple mathematical model for a biological neuron. A perceptron receives input from other neurons and it applies a linear transformation to the input, followed by a non-linear activation function.

Deep learning networks can be feedforward networks where the data flows in one direction from input to output, or recurrent networks where the data can flow in a cyclic fashion.

There are different types of deep learning architectures such as feedforward neural networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs) and Generative Adversarial Networks (GANs). These architectures are designed to learn specific types of features and patterns from different types of data.

Deep learning models are trained using large amounts of labeled data, and typically use supervised or semi-supervised learning techniques. The training process involves adjusting the weights of the network to minimize a loss function, which measures the difference between the predicted output and the true output. This process is known as back-propagation, which is an algorithm for training the weights in a neural network by propagating the error back through the network. In the first AI book I wrote in the 1980s I covered the implementation of back-propagation in detail. As I write the material here on deep learning I think that it is more important for you to have the skills to choose appropriate tools for different applications and be less concerned about low-level implementation details. I think this characterizes the change in trajectory of AI from being about tool building to the skills of using available tools and sometimes previously trained models while spending more of your effort analyzing business functions and in general application domains.

Deep Learning has been applied to various fields such as Computer Vision, Natural Language Processing, Speech Recognition, etc.

Using TensorFlow and Keras for Building a Cancer Prediction Model

Please follow this link to Google Colab[*] to see the example using TensorFlow to build a model of the University of Wisconsin cancer dataset. A subset of this Jupyter notebook can also be found in the file **deep-learning/wisconsin_data_github.py** but you will need to install all dependencies automatically installed by Colab and you might need to remove the calls to TensorBoard.

We use the Python package skimpy† that is a lightweight tool for creating summary statistics from Pandas‡ dataframes. Please note that I will use Pandas dataframes without much explaination so if you have never used Pandas please review the tutorial on importing and using CSV spreadsheet data§. The other tutorials are optional.

We use the **skimpy** library to get information on our dataset:

```
train_uri = "https://raw.githubusercontent.com/mark-watson/cancer-deep-learning-mode\
l/master/train.csv"
test_uri = "https://raw.githubusercontent.com/mark-watson/cancer-deep-learning-model\
/master/test.csv"

train_df = pandas.read_csv(train_uri, header=None)

skim(train_df)
```

[*]https://colab.research.google.com/drive/18UJ-5i6_SyfU01PptfNxvR3ZCNirrRgc?usp=sharing
†https://pypi.org/project/skimpy/
‡https://pandas.pydata.org
§https://pandas.pydata.org/docs/getting_started/intro_tutorials/02_read_write.html#min-tut-02-read-write

skimpy summary

Data Summary

dataframe	Values
Number of rows	497
Number of columns	10

Data Types

Column Type	Count
int64	10

number

	missing	complete rate	mean	sd	p0	p25	p75	p100	hist
0	0	1	4.5	2.8	1	2	6	10	
1	0	1	3.2	3.1	1	1	5	10	
2	0	1	3.3	3	1	1	5	10	
3	0	1	2.9	2.9	1	1	4	10	
4	0	1	3.2	2.2	1	2	4	10	
5	0	1	3.6	3.6	1	1	6	10	
6	0	1	3.4	2.5	1	2	5	10	
7	0	1	2.9	3.1	1	1	4	10	
8	0	1	1.6	1.7	1	1	1	10	
9	0	1	0.36	0.48	0	0	1	1	

End

We need to prepare the training and test data:

```
train = train_df.values
X_train = train[:,0:9].astype(float) # 9 inputs
print("Number training examples:", len(X_train))
Y_train = train[:,-1].astype(float)  # one target output (0 for no cancer, 1 for mal\
ignant)
test = pandas.read_csv(test_uri, header=None).values
X_test = test[:,0:9].astype(float)
Y_test = test[:,-1].astype(float)
```

We now need to define the TensorFlow/Keras model architecture and train the model:

```
model = Sequential()
model.add(Dense(tf.constant(15), input_dim=tf.constant(9), activation='relu'))
model.add(Dense(tf.constant(15), input_dim=tf.constant(15), activation='relu'))
model.add(Dropout(0.2)),
model.add(Dense(tf.constant(1), activation='sigmoid'))
model.summary()

model.compile(optimizer='sgd',
              loss='mse',
              metrics=['accuracy'])

logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
callbacks = [TensorBoard(log_dir=logdir,histogram_freq=1,write_graph=True, write_ima\
```

```
ges=True)]

model.fit(X_train, Y_train, batch_size=100, epochs=60, callbacks=callbacks)
```

We use the trained model to make predictions on test data:

```
y_predict = model.predict([[4,1,1,3,2,1,3,1,1], [3,7,7,4,4,9,4,8,1]])
print("* y_predict (should be close to [[0], [1]]):", y_predict)

* y_predict (should be close to [[0], [1]]): [[0.37185097]
 [0.9584093 ]]
```

You can compare this example using TensorFlow and Keras to our similar classification example using the same data where we used the **Scikit-learn** library.

PyTorch and JAX

In addition to TensorFlow/Keras, the two other most popular frameworks for deep learning are PyTorch* and JAX†.

All three frameworks are popular and well supported. I started studying deep learning at the same time that TensorFlow was initially released and I use TensorFlow (usually with the easier to use Keras APIs) for at least 90% of my professional deep learning work. Because of my own history I am showing you TensorFlow/Keras examples but if PyTorch or JAX appeal more to you then by all means use the framework that fits your requirements.

*https://pytorch.org/tutorials/beginner/basics/intro.html
†https://jax.readthedocs.io/en/latest/notebooks/quickstart.html

Natural Language Processing Using Deep Learning

I spent several years in the 1980s using symbolic AI approaches to Natural Language Processing (NLP) like augmented transition networks and conceptual dependency theory with mixed results. For small vocabularies and small domains of discourse these techniques yielded modestly successful results. I now only use Deep Learning approaches to NLP in my work.

Deep Learning in NLP is a branch of machine learning that utilizes deep neural networks to understand, interpret and generate human language. It has revolutionized the field of NLP by improving the accuracy of various NLP tasks such as text classification, language translation, sentiment analysis, and natural language generation (e.g., ChatGPT).

Deep learning models such as Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and Transformer models have been used to achieve state-of-the-art performance on various NLP tasks. These models have been trained on large amounts of text data, which has allowed them to learn complex patterns in human language and improve their understanding of the context and meaning of words.

The use of pre-trained models, such as BERT and GPT-3, has also become popular in NLP and I use both frequently for my work. These models have been pre-trained on a large corpus of text data, and can be fine-tuned for a specific task, which significantly reduces the amount of data and computing resources required to train a derived model.

Deep learning in NLP has been applied in various industries such as chatbots, automated customer service, and language translation services. It has also been used in research areas such as natural language understanding, question answering, and text summarization.

As a result, Deep Learning in NLP has greatly improved the performance of various NLP tasks by utilizing deep neural networks to understand and interpret human language. The use of pre-trained models has also made it easier to fine-tune models for specific tasks, which has led to a wide range of applications in industry and research.

In the last decade deep learning techniques have solved most NLP problems, at least in a "good enough" engineering sense. As I write this the ChatGPT model has scored 80% accuracy on the verbal SAT college admissions test. In this chapter we will experiment with a few useful public models that can be used as paid for API calls or in many cases you can run the models yourself.

OpenAI GPT-3 APIs

OpenAI GPT-3 (Generative Pre-trained Transformer 3) is an advanced language processing model developed by OpenAI. There are three general classes of OpenAI API services:

- GPT-3 which performs a variety of natural language tasks.
- Codex which translates natural language to code.
- DALL·E which creates and edits original images.

GPT-3 is capable of generating human-like text, completing tasks such as language translation, summarization, and question answering, and much more. OpenAI offers GPT-3 APIs, which allow developers to easily integrate GPT-3's capabilities into their applications.

The GPT-3 API provides a simple and flexible interface for developers to access GPT-3's capabilities such as text completion, language translation, and text generation. The API can be accessed using a simple API call, and can be integrated into a wide range of applications such as chatbots, language translation services, and text summarization.

Additionally, OpenAI provides a number of pre-built models that developers can use, such as the GPT-3 language model, the GPT-3 translation model, and the GPT-3 summarization model. These pre-built models allow developers to quickly and easily access GPT-3's capabilities without the need to train their own models.

Overall, the OpenAI GPT-3 APIs provide a powerful and easy-to-use tool for developers to integrate advanced language processing capabilities into their applications, and can be a game changer for developers looking to add natural language processing capabilities to their projects.

We will only use the GPT-3 APIs here. The following examples are derived from the official set of cookbook examples at https://github.com/openai/openai-cookbook. The first example calls the OpenAI GPT-3 Completion API with a sample of input text and the model completes the text (deep-learning/openai/openai-example.py):

```python
# from OpenAI's documentation

import os

import openai

openai.api_key = os.environ.get('OPENAI_KEY')

# create a completion
input_text = "Mary went to the grocery store. She bought"
completion = openai.Completion.create(engine="davinci",
                                      prompt=input_text)

# print the completion
print(completion.choices[0].text)
```

Every time you run this example you get different output. Here is one example run:

```
$ python openai-example.py
 bread, butter, and tomatoes. She ran into some old friends, but she
```

Using GPT-3 to Name K-Means Clusters

I get a lot of enjoyment finding simple application examples that solve problems that I had previously spent a lot of time solving with other techniques. As an example, around 2010 a customer and I created some ad hoc ways to name K-Means clusters with meaningful cluster names. Several years later at Capital One, my team brainstormed other techniques for assigning meaningful cluster names for the patent SYSTEM TO LABEL K-MEANS CLUSTERS WITH HUMAN UNDERSTANDABLE LABELS*.

One of the OpenAI example Jupyter notebooks https://github.com/openai/openai-cookbook/blob/main/examples/Clustering.ipynb solves this problem elegantly using a text prompt like:

```
f'What do the following customer reviews have in common?\n\nCustomer reviews:\n"""\n\
{reviews}\n"""\n\nTheme:'
```

where the variable **reviews** contains the concatenated recipe reviews in a specific cluster. The recipe clusters are named like:

```
Cluster 0 Theme:  All of the reviews are positive and the customers are satisfied wi\
th the product they purchased.

Cluster 1 Theme:  All of the reviews are about pet food.

Cluster 2 Theme:  All of the reviews are positive and express satisfaction with the \
product.

Cluster 3 Theme:  All of the reviews are about food or drink products.
```

Using GPT-3 to Translate Natural Language Queries to SQL

Another example of a long term project I had that is now easily solved with the OpenAI GPT-3 models is translating natural language queries to SQL queries. I had an example I wrote for the first two editions of my Java AI book† (I later removed this example because the code was difficult to follow). I later reworked this example in Common Lisp and used both versions in several consulting projects in the late 1990s and early 2000s.

I refer you to one of the official OpenAI examples https://github.com/openai/openai-cookbook/blob/main/examples/Backtranslation_of_SQL_queries.py. In my Java and Common Lisp NLP query

*https://patents.justia.com/patent/20210357429
†https://leanpub.com/javaai

examples, I would test generated SQL queries against a database to ensure they were legal queries, etc., and if you modify OpenAI's example I suggest you do the same.

Here is a sample query and a definition of available database tables:

```
nl_query: str = "Return the name of each department that had more than 10 employees \
in June 2021",
eval_template: str = "{};\n-- Explanation of the above query in human readable forma\
t\n-- {}",
table_definitions: str = "# Employee(id, name, department_id)\n# Department(id, name\
, address)\n# Salary_Payments(id, employee_id, amount, date)\n",
```

Here is the output to OpenAI's example:

```
$ python Backtranslation_of_SQL_queries.py
SELECT department.name
FROM department
JOIN employee ON department.id = employee.department_id
JOIN salary_payments ON employee.id = salary_payments.employee_id
WHERE salary_payments.date BETWEEN '2021-06-01' AND '2021-06-30'
GROUP BY department.name
HAVING COUNT(*) > 10
```

I find this to be a great example of creatively using deep learning via pre-trained models. I urge you, dear reader, to take some time to peruse the Hugging Face example Jupyter notebooks to see which might be applicable to your development projects. I have always felt that my work "stood on the shoulders of giants," that is my work builds on that of others. In the new era of deep learning and large language models, where very large teams work on models and technology that individuals can't compete with, being a solo developer or when working for a small company, we need to be flexible and creative in using resources from OpenAI, Hugging Face, Google, Microsoft, Meta, etc.

Hugging Face APIs

Hugging Face provides an extensive library of pre-trained models and a set of easy-to-use APIs that allow developers to quickly and easily integrate NLP capabilities into their applications. The pre-trained models are based on the state-of-the-art transformer architectures, which have been trained on large corpus of data and can be fine-tuned for specific tasks, making it easy for developers to add NLP capabilities to their projects. Hugging Face maintains a task page listing all kinds of machine learning that they support https://huggingface.co/tasks for task domains:

- Computer Vision
- Natural Language Processing

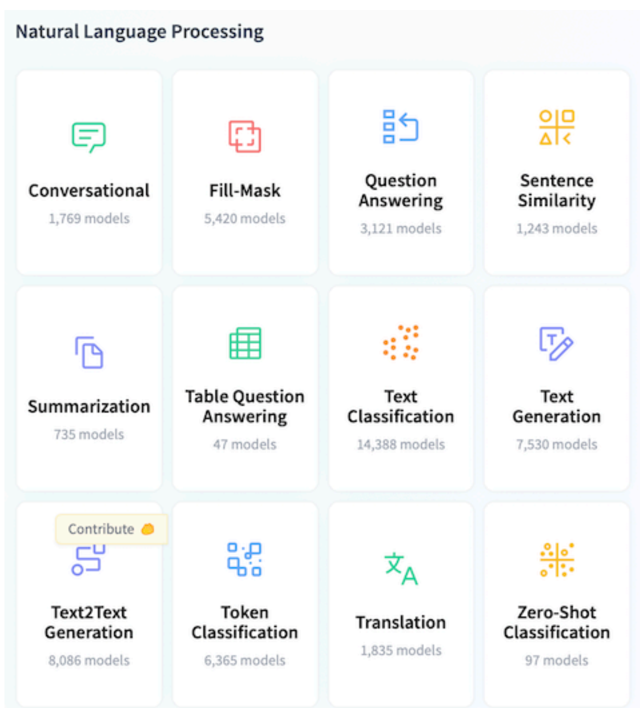
- Audio
- Tabular Data
- Multimodal
- Reinforcement Learning

As an open source and open model company, Hugging Face is a provider of NLP technology, with a focus on developing and providing state-of-the-art pre-trained models and tools for NLP tasks. They have developed a library of pre-trained models, including models based on transformer architectures such as BERT, GPT-2, and GPT-3, which can be fine-tuned for various tasks such as language understanding, language translation, and text generation.

Hugging Face also provides a set of APIs, which allows developers to easily access the functionality of these pre-trained models. The APIs provide a simple and flexible interface for developers to access the functionality of these models, such as text completion, language translation, and text generation. This allows developers to quickly and easily integrate NLP capabilities into their applications, without the need for extensive knowledge of NLP or deep learning.

The Hugging Face APIs are available via a simple API call and are accessible via an API key. They support a wide range of programming languages such as Python, Java and JavaScript, making it easy for developers to integrate them into their application.

Since my personal interests are mostly in Natural Language Processing (NLP) as used for processing text data, automatic extraction of structured data from text, and question answering systems, I will just list their NLP task types:



Coreference: Resolve Pronouns to Proper Nouns in Text Using Hugging Face APIs

You can find this example script in **PythonPracticalAIBookCode/deep-learning/huggingface_-apis/hf-coreference.py**:

```python
import json
import requests
import os
from pprint import pprint

# NOTE: Hugging Face does not have a direct anaphora resolution model, so this
#       example is faking it using masking with a BERT model.

HF_API_TOKEN = os.environ.get('HF_API_TOKEN')
headers = {"Authorization": f"Bearer {HF_API_TOKEN}"}
API_URL = "https://api-inference.huggingface.co/models/bert-base-uncased"

def query(payload):
    data = json.dumps(payload)
    response = requests.request("POST", API_URL, headers=headers, data=data)
    return json.loads(response.content.decode("utf-8"))

data = query("John Smith bought a car. [MASK] drives it fast.")

pprint(data)
```

Here is example output (I am only showing the highest scored results for each query):

```
$ python hf-coreference.py
[{'score': 0.9037206768989563,
  'sequence': 'john smith bought a car. he drives it fast.',
  'token': 2002,
  'token_str': 'he'},
 {'score': 0.015135547146201134,
  'sequence': 'john smith bought a car. john drives it fast.',
  'token': 2198,
  'token_str': 'john'}]
```

GPT-2 Hugging Face Documentation

The documentation page for Hugging Face GPT2 models* has many examples for using their GPT2 model for tokenization and other NLP tasks.

Answer Questions From Text

We already saw an example Jupyter Notebook in the chapter *Semantic Web, Linked Data and Knowledge Graphs* in the section *A Hybrid Deep Learning and RDF/SPARQL Application for Question Answering* that used the the Hugging Face **NeuML/bert-small-cord19-squad2** model to use a large context text sample to answer questions.

Calculate Semantic Similarity of Sentences Using Hugging Face APIs

Given a list of sentences we, can calculate sentence embeddings for each one. Any new sentence can be matched by calculating its embedding and finding the closest cosine similarity match. Contents of file **hf-sentence_similarities.py**:

```
import json
import requests
import os
from pprint import pprint

HF_API_TOKEN = os.environ.get('HF_API_TOKEN')
headers = {"Authorization": f"Bearer {HF_API_TOKEN}"}
API_URL = "https://api-inference.huggingface.co/models/sentence-transformers/all-Min\
iLM-L6-v2"

def query(payload):
    data = json.dumps(payload)
    response = requests.request("POST", API_URL, headers=headers, data=data)
    return json.loads(response.content.decode("utf-8"))
data = query(
    {
        "inputs": {
            "source_sentence": "That is a happy person",
            "sentences": ["That is a happy dog", "That is a very happy person", "Tod\
ay is a sunny day"],
        }
```

*https://huggingface.co/docs/transformers/model_doc/gpt2

```
    }
)
pprint(data)
```

Here is example output:

```
$ python hf-sentence_similarities.py
[0.6945773363113403, 0.9429150819778442, 0.2568760812282562]
```

Here we are using one of the free Hugging Face APIs. At the end of this chapter we will use an alternative sentence embedding model that you can easily run on your laptop.

Summarizing Text Using a Pre-trained Hugging Face Model on Your Laptiop

For most Hugging Face pre-trained models you can either use them running on Hugging Face servers via an API call or use the **transformers** library to download and run the model on your laptop. The downloaded model and associated files are a little less than two gigabytes of data. Once a model is downloaded to **~/.cache/huggingface** on your local filesystem you can use the model again without re-downloading it.

```
from transformers import pipeline
from pprint import pprint

summarizer = pipeline("summarization", model="facebook/bart-large-cnn")

text = "The President sent a request for changing the debt ceiling to Congress. The \
president might call a press conference. The Congress was not oblivious of what the
Supreme Court's majority had ruled on budget matters. Even four Justices had found n
othing to criticize in the President's requirement that the Federal Government's fou
r-year spending plan. It is unclear whether or not the President and Congress can co
me to an agreement before Congress recesses for a holiday. There is major dissagreme
nt between the Democratic and Republican parties on spending."

results = summarizer(text, max_length=60)[0]
print("".join(results['summary_text']))
```

Here is some sample output:

```
$ python hf-summarization.py
The President sent a request for changing the debt ceiling to Congress. The Congress\
 was not oblivious of what the Supreme Court's majority had ruled on budget matters.
 Even four Justices had found nothing to criticize in the President's requirement th
at the Federal Government's four-year spending plan be changed
```

Zero Shot Classification Using Hugging Face APIs

Zero shot classification models work by specifying which classification labels you want to assign to input texts. In this example we ask the model to classify text into one of: '**refund**', '**faq**', '**legal**'. Usually for text classification applications we need to pre-train a model using examples of text in different categories we are interested in. This extra work is not required using the Meta/Facebook pre-trained zero shot model **bart-large-mnli**:

```
import json
import requests
import os
from pprint import pprint

HF_API_TOKEN = os.environ.get('HF_API_TOKEN')
headers = {"Authorization": f"Bearer {HF_API_TOKEN}"}
API_URL = "https://api-inference.huggingface.co/models/facebook/bart-large-mnli"

def query(payload):
    data = json.dumps(payload)
    response = requests.request("POST", API_URL, headers=headers, data=data)
    return json.loads(response.content.decode("utf-8"))
data = query(
    {
        "inputs": "Hi, I recently bought a device from your company but it is not wo\
rking as advertised and I would like to get reimbursed!",
        "parameters": {"candidate_labels": ["refund", "legal", "faq"]},
    }
)
pprint(data)
```

Here is some example output:

```
$ python hf-zero_shot_classification.py
{'labels': ['refund', 'faq', 'legal'],
 'scores': [0.877787709236145, 0.10522633045911789, 0.01698593981564045],
 'sequence': 'Hi, I recently bought a device from your company but it is not '
             'working as advertised and I would like to get reimbursed!'}
```

Comparing Sentences for Similarity Using Transformer Models

Although I use OpenAI and HuggingFace for most of the pre-trained NLP models I use, I recently used a sentence similarity Transformer model from the Ubiquitous Knowledge Processing Lab* for a quick work project and their library support for finding similar sentences is simple to use. This model was written using PyTorch. Here is one of their examples, slightly modified for this book:

```
# pip install sentence_transformers
# The first time this script is run, the sentence_transformers library will
# download a pre-trained model.

# This example is derived from examples at https://www.sbert.net/docs/quickstart.html

from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer('all-MiniLM-L6-v2')

sentences = ['The IRS has new tax laws.',
             'Congress debating the economy.',
             'The polition fled to South America.',
             'Canada and the US will be in the playoffs.',
             'The cat ran up the tree',
             'The meal tasted good but was expensive and perhaps not worth the price\
.']

#Sentences are encoded by calling model.encode()
sentence_embeddings = model.encode(sentences)

#Compute cosine similarity between all pairs
cos_sim = util.cos_sim(sentence_embeddings, sentence_embeddings)

#Add all pairs to a list with their cosine similarity score
all_sentence_combinations = []
```

*https://www.informatik.tu-darmstadt.de/ukp/ukp_home/index.en.jsp

```
for i in range(len(cos_sim)-1):
    for j in range(i+1, len(cos_sim)):
        all_sentence_combinations.append([cos_sim[i][j], i, j])

#Sort list by the highest cosine similarity score
all_sentence_combinations =
  sorted(all_sentence_combinations, key=lambda x: x[0], reverse=True)

print("Top-8 most similar pairs:")
for score, i, j in all_sentence_combinations[0:8]:
    print("{} \t {} \t {:.4f}".format(sentences[i],
                                      sentences[j],
                                      cos_sim[i][j]))
```

The output is:

```
$ python sentence_transformer.py
Top-8 most similar pairs:
The IRS has new tax laws.    Congress debating the economy.      0.1793
Congress debating the economy.   Canada and the US will be in the playoffs.      0.1\
210
Congress debating the economy.   The meal tasted good but was expensive and perhaps \
not worth the price.      0.1131
Congress debating the economy.   The polition fled to South America.     0.0963
The polition fled to South America.      Canada and the US will be in the playoffs. \
     0.0854
The polition fled to South America.      The meal tasted good but was expensive and \
perhaps not worth the price.     0.0826
The polition fled to South America.      The cat ran up the tree     0.0809
Congress debating the economy.   The cat ran up the tree     0.0496
```

A common use case might be a customer service facing chatbot where we simply match the user's question with all recorded user questions that have accepted "canned answers." The runtime to get the best match is **O(N)** where **N** is the number of previously recorded user questions. The cosine similarity calculation, given two embedding vectors, is very fast.

In this example we used the Sentence Transformer utility library util.py[*] to calculate the cosine similarities between all combinations of sentence embeddings. For a practical application you can use the **cos_sim** function in **util.py**:

[*]https://github.com/UKPLab/sentence-transformers/blob/master/sentence_transformers/util.py

```
>>> from util
>>> util.cos_sim(sentence_embeddings[0], sentence_embeddings[1])
tensor([[0.1793]])
>>>
```

Deep Learning Natural Language Processing Wrap-up

In this example and earlier chapters we have seen examples of how effective deep learning is for NLP. I worked on other methods of NLP over a 25-year period and I ask you, dear reader, to take my word on this: deep learning has revolutionized NLP and for almost all practical NLP applications deep learning libraries and models from organizations like Hugging Face and OpenAI should be the first thing that you consider using.

Part IV - Overviews of Image Generation, Reinforcement Learning, and Recommendation Systems

This final part of this book consists of overviews of three important topics that I cover briefly, with perhaps more material added in the next edition of this book.

Overview of Image Generation

I have never used deep learning image generation at work but I have fun experimenting with both code and model examples, as well as turn-key web apps like DALL·E. We will use Brett Kuprel's Mini-Dalle model* GitHub repository that is a reduced size port of DALL·E Mini to PyTorch.

You can run this example directly on Google Colab†. Here is a listing of the example code in this notebook:

```
!pip install min-dalle

import os

from IPython.display import display, update_display
import torch
from min_dalle import MinDalle

dtype = "float32"

model = MinDalle(
    dtype=getattr(torch, dtype),
    device='cuda',
    is_mega=True,
    is_reusable=True
)

directory = "/content"

for root, subdirectories, files in os.walk(directory):

    for filename in files:
        if filename.endswith(".png"):
          path_img = os.path.join(root, filename)
          os.remove(path_img)

text = "parrot sitting on old man's shoulder"

image_stream = model.generate_image_stream(
```

*https://github.com/kuprel/min-dalle
†https://colab.research.google.com/drive/1FxTaCCVtLWUfvHKvcgnwAerJtq5a6KSX?usp=sharing

```
30      text=text,
31      seed=-123,
32      grid_size=2,
33      progressive_outputs=True,
34      is_seamless=False,
35      temperature=1.5,
36      top_k=int(256),
37      supercondition_factor=float(12)
38  )
39
40  for image in image_stream:
41      display(image, display_id=1)
42      # optional:
43      image.save("./"+text.replace(" ", "_")+".png")
```

The pre-trained model files will be downloaded the first time you run this code. We create a class instance in lines 11-16. If **is_mega** is true then a larger model is constructed. If **is_reusable** is true then the same model is reused to create additional images.

The example prompt text "parrot sitting on old man's shoulder" set in line 27 can be changed to whatever you want to try.

You can try changing the temperature (increase for more randomness and differences from training examples), random seed, and text prompt. This is a generated image containing four images (because we set the output image grid size to 2):

I reduced the above image size by a factor of four in order to keep the size of this eBook fairly small. When you run this example you will get higher resolution images.

You will get different results even without changing the random seed or parameters. Here is sample output from the second time I ran this example on Google Colab:

I also reduced the last image size by a factor of four for inclusion in this chapter.

The three Python model files in the GitHub repository comprise about 600 lines of code making this a fairly short complete Attention Network/Transformer example. We will not walk through the code here but if your are interested in the implementation please read the original paper from Open AI Zero-Shot Text-to-Image Generation* before reading the code for the models†.

Recommended Reading for Image Generation

The example program is small enough to run on Google Colab or on your laptop (you may want to reduce the value **top_k=int(256)** to 128 if you are not using a GPU with 16G of video RAM.

You can get more information on DALL·E and DALL·E 2 from https://openai.com/blog/dall-e/. You will get much higher quality images using OpenAI's DALL·E web service.

We won't cover StyleGAN (created by researchers at NVIDIA) here because it is almost two year old technology as I am writing this chapter but I recommend experimenting with it using

*https://arxiv.org/abs/2102.12092
†https://github.com/kuprel/min-dalle/tree/main/min_dalle/models

the TensorFlow/Keras StyleGAN example*. StyleGAN can progressively increase the resolution of images. StyleGAN can also mix styles from multiple images to create a new image.

*https://keras.io/examples/generative/stylegan/

Overview of Reinforcement Learning (Optional Material)

Reinforcement Learning has been used in various applications such as robotics, game playing, recommendation systems, etc. Reinforcement Learning (RL) is a broad topic and we will only cover aspects RL that I use myself.

This is a common theme in this book: if I don't love a topic or I don't have much practical experience with it, I generally don't write about it or cover it lightly with references for further study. I have limited experience using RL professionally, mostly for a project a few years ago at Capital One and here I am guiding you on the same learning path that I took prior to working on that project.

Overview

Reinforcement Learning is a type of machine learning that is concerned with decision-making in dynamic and uncertain environments. RL uses the concept of an agent which interacts with its environment by taking actions and receiving feedback in the form of rewards or penalties. The goal of the agent is to learn a policy which is a mapping from states of the environment to actions with the goal of maximizing the expected cumulative reward over time.

There are several key components to RL:

- Environment: the system or "world" that the agent interacts with.
- Agent: the decision-maker that chooses actions based on its current state, the current environment, and its policy.
- State: a representation of the current environment, the parameters and trained policy of the agent, and possibly the visible actions of other agents in the environment.
- Action: a decision taken by the agent.
- Reward: a scalar value that the agent receives as feedback for its actions.

Reinforcement learning algorithms can be divided into two main categories: value-based and policy-based. In value-based RL the agent learns an estimate of the value of different states or state-action pairs which are then used to determine the optimal policy. In contrast, in policy-based RL the agent directly learns a policy without estimating the value of states or state-action pairs.

Reinforcement Learning can be implemented using different techniques such as Q-learning, SARSA, DDPG, A2C, PPO, etc. Some of these techniques are model-based, which means that the agent uses a model of the environment to simulate the effects of different actions and plan ahead. Others are

model-free, which means that the agent learns directly from the rewards and transitions experienced in the environment.

If you enjoy the overview material in this chapter I recommend that you consider investing the time in the Coursera RL specialization https://www.coursera.org/learn/fundamentals-of-reinforcement-learning* taught by Martha and Adam White. There are 50+ RL courses on Coursera†. I took the courses taught by Martha and Adam White before starting my RL project at Capital One.

My favorite RL book is "Reinforcement Learning: An Introduction, second edition" by Richard Sutton and Andrew Barto, that can be read online for free at http://www.incompleteideas.net/book/the-book-2nd.html. They originally wrote their book examples in Common Lisp but most of the code is available rewritten in Python. The Common Lisp code for the examples is here‡. Shangtong Zhang translated the book examples to Python, available here§. Martha and Adam White's Coursera class uses this book as a reference.

The core idea of RL is that we train a software agent to interact with and change its environment based on its expectations of the utility of current actions improving metrics for success in the future. There is some tension between writing agents that simply reuse past actions that proved to be useful, rather than aggressively exploring new actions in the environment. There are interesting formalisms for this that we will cover.

There are two general approaches to providing training environments to Reinforcement Learning trained agents: physically devices in the real world or simulated environments. This is not a book on robotics so we use the second option.

The end goal for modeling a RL problem is calculating a policy that can be used to control an agent in environments that are similar to the training environment. In a model at time **t** we have a given $State_t$. RL policies can be continually be updated during training and in production environments. A policy given a $State_t$, calculates an $Action_t$ to execute and changes the state to $State_{t+1}$.

Available RL Tools

For initial experiments with RL, I would recommend taking the same path that I took before using RL at work:

- Using a maintained fork of OpenAI's Gym library Gymnasium¶.
- Taking the Coursera classes by Martha and Adam White.
- The Sutton/Barto RL Book and accompanying Common Lisp or Python examples.

The original OpenAI RL Gym was a good environment for getting started with simple environments and examples but I didn't get very far with self-study. The RL Coursera classes were a great overview

*https://www.coursera.org/learn/fundamentals-of-reinforcement-learning#instructors
†https://www.coursera.org/courses?query=reinforcement%20learning
‡http://www.incompleteideas.net/book/code/code2nd.html
§https://github.com/ShangtongZhang/reinforcement-learning-an-introduction
¶https://github.com/Farama-Foundation/Gymnasium

of theory and practice, and I then spend as much time as I could spare working through Sutton/Barto before my project started.

Reinforcement Learning Wrap-up

Dear reader, please pardon the brevity of this overview chapter. I may re-work this chapter with a few examples in the next edition of this book. I tagged this chapter as optional material because I feel that most readers will be better off investing limited learning time in understanding how to use deep learning and pre-trained models.

Overview of Recommendation Systems

Recommendation systems are a type of information filtering system that utilize historical data, such as past user behavior or interactions, to predict the likelihood of a user's interest in certain items or products. As an example application: if a product web site has 100K products that is too many for customers to browse through. Based on a customers past purchases, finding customers with similar purchases, etc. it is possible to filter the products shown to a customer.

Writing recommendation systems is a common requirement for almost all businesses that sell products to customers. Before we get started we need to define two terms that you may not be familiar with: Collaborative filtering*: uses both similarities between users and items to calculate recommendations. This linked Wikipedia article also discusses content-based filtering which uses user and item features.

The Movie Lens dataset† created by the GroupLens Research organization uses the user movie preference https://movielens.org dataset. This dataset is a standard for developing and evaluating recommendation system algorithms and models.

There are at least three good approaches to take:

- Use a turnkey recommendation system like Amazon Personalize‡ that is a turn-key service on AWS. You can evaluate Amazon Personalize for your company's use by spending about one hour working through the getting started tutorial§.
- Use one of the standard libraries or TensorFlow implementations for the classic approach using [Matrix Factorization](https://en.wikipedia.org/wiki/Matrix_factorization_(recommender_systems) for collaborative filtering. Examples are Eric Lundquist's Python library rankfm¶ and the first example for the TensorFlow Recommenders library‖. Google has a good Matrix Factorization tutorial**. While I prefer using the much more complicated TensorFlow Recommenders library, using matrix factorization is probably a good way to start and I recommend taking an hour to work through this Google Colab tutorial††
- Use the TensorFlow Recommenders‡‡ library that supports multi-tower deep learning models that use data for user interactions, user detail data, and product detail data.

*https://en.wikipedia.org/wiki/Collaborative_filtering
†https://grouplens.org/datasets/movielens/
‡https://aws.amazon.com/personalize/
§https://github.com/aws-samples/amazon-personalize-samples
¶https://github.com/etlundquist/rankfm
‖https://www.tensorflow.org/recommenders/examples/quickstart
**https://developers.google.com/machine-learning/recommendation/collaborative/matrix
††https://colab.research.google.com/github/google/eng-edu/blob/main/ml/recommendation-systems/recommendation-systems.ipynb
‡‡https://www.tensorflow.org/recommenders

We will not write any recommendation systems from scratch in this chapter. We will review one open source recommendation system that I have used at work, the TensorFlow Recommenders library.

Recommendation systems can use a wide variety of techniques, such as collaborative filtering, content-based filtering, and hybrid methods combining filtering algorithms, Matrix Factorization, or Deep Learning technologies, etc. to generate personalized recommendations for users. Collaborative filtering algorithms make recommendations based on the actions of similar users, while content-based filtering algorithms base recommendations on the attributes of items that a user has previously shown interest in. Hybrid methods may be further enhanced by incorporating additional data sources, such as demographic information, or by utilizing more advanced machine learning techniques, such as deep learning or reinforcement learning.

TensorFlow Recommenders

I used Google's TensorFlow Recommenders library for a work project. I recommend it because it has very good documentation, many examples using the Movie Lens dataset, and is fairly easy to adapt to general user/product recommendation systems.

We will refer to the documentation and examples https://www.tensorflow.org/recommenders and follow the last Movie Lens example.

There are several types of data that could be used for recommending movies:

- User interactions (selecting movies).
- User data. User data is not used in this example, but for a product recommendation system I have created embeddings of all available data features associated with users.
- Movie data based on text embedding of movie titles. Note: for product recommendation systems you might still use text embedding of product descriptions, but you would also likely create embeddings based on product features.

We will use the TensorFlow Recommenders using rich features example*. For the following overview discussion, you may want to either open this link to read this example or open the alternative Google Colab example link† to run the example on Colab. Please note that this example takes about ten minutes to run on Colab. For our discussion, I will use short code snippets and use one screenshot of the example in Colab so you can optionally just follow along without opening either link for now.

The TF recommenders example starts with reading the Movie Lens‡ dataset using the TensorFlow Data library:

*https://www.tensorflow.org/recommenders/examples/deep_recommenders
†https://colab.research.google.com/github/tensorflow/recommenders/blob/main/docs/examples/deep_recommenders.ipynb
‡https://grouplens.org/datasets/movielens/

```
import tensorflow_datasets as tfds
ratings = tfds.load("movielens/100k-ratings", split="train")
movies = tfds.load("movielens/100k-movies", split="train")

ratings = ratings.map(lambda x: {
    "movie_title": x["movie_title"],
    "user_id": x["user_id"],
    "timestamp": x["timestamp"],
})
movies = movies.map(lambda x: x["movie_title"])
```

We need to later generate embedding layers for both unique movie titles and also unique user IDs. We start with getting sequences for unique movie titles and user IDs:

```
unique_movie_titles = np.unique(np.concatenate(list(movies.batch(1000))))
unique_user_ids = np.unique(np.concatenate(list(ratings.batch(1_000).map(
    lambda x: x["user_id"]))))
```

In this example Python user and movie models are derived from the Python class **tf.keras.Model**. Let's look at the implementation of these two models:

```
class UserModel(tf.keras.Model):

  def __init__(self):
    super().__init__()

    self.user_embedding = tf.keras.Sequential([
        tf.keras.layers.StringLookup(
            vocabulary=unique_user_ids, mask_token=None),
        tf.keras.layers.Embedding(len(unique_user_ids) + 1, 32),
    ])
    self.timestamp_embedding = tf.keras.Sequential([
        tf.keras.layers.Discretization(timestamp_buckets.tolist()),
        tf.keras.layers.Embedding(len(timestamp_buckets) + 1, 32),
    ])
    self.normalized_timestamp = tf.keras.layers.Normalization(
        axis=None
    )

    self.normalized_timestamp.adapt(timestamps)

  def call(self, inputs):
```

```
    # Take the input dictionary, pass it through each input layer,
    # and concatenate the result.
    return tf.concat([
        self.user_embedding(inputs["user_id"]),
        self.timestamp_embedding(inputs["timestamp"]),
        tf.reshape(self.normalized_timestamp(inputs["timestamp"]), (-1, 1)),
    ], axis=1)
```

The function **tf.keras.layers.StringLookup** is used to create an embedding layer from a sequence of unique string IDs. Timestamps for user selection events are fairly continuous so we use **tf.keras.layers.Discretization** to collapse a wide range of timestamp values into discrete bins.

Classed derived from class **tf.keras.Model** are expected to implement a **call** method that is passed a inputs and returns a single Tensor of concatenated inputs and timestamp embeddings.

We build a similar model for movies:

```
class MovieModel(tf.keras.Model):

  def __init__(self):
    super().__init__()

    max_tokens = 10_000

    self.title_embedding = tf.keras.Sequential([
      tf.keras.layers.StringLookup(
          vocabulary=unique_movie_titles,mask_token=None),
      tf.keras.layers.Embedding(len(unique_movie_titles) + 1, 32)
    ])

    self.title_vectorizer = tf.keras.layers.TextVectorization(
        max_tokens=max_tokens)

    self.title_text_embedding = tf.keras.Sequential([
      self.title_vectorizer,
      tf.keras.layers.Embedding(max_tokens, 32, mask_zero=True),
      tf.keras.layers.GlobalAveragePooling1D(),
    ])

    self.title_vectorizer.adapt(movies)

  def call(self, titles):
    return tf.concat([
        self.title_embedding(titles),
```

```
      self.title_text_embedding(titles),
  ], axis=1)
```

The class **MovieModel** is different than the class **UserModel** since we create embeddings for movie titles instead of IDs.

We also wrap the user model in a separate query model that combines a user model with dense fully connected layers:

```
class QueryModel(tf.keras.Model):
  """Model for encoding user queries."""

  def __init__(self, layer_sizes):
    """Model for encoding user queries.

    Args:
      layer_sizes:
        A list of integers where the i-th entry represents the number of units
        the i-th layer contains.
    """
    super().__init__()

    # We first use the user model for generating embeddings.
    self.embedding_model = UserModel()

    # Then construct the layers.
    self.dense_layers = tf.keras.Sequential()

    # Use the ReLU activation for all but the last layer.
    for layer_size in layer_sizes[:-1]:
      self.dense_layers.add(tf.keras.layers.Dense(layer_size, activation="relu"))

    # No activation for the last layer.
    for layer_size in layer_sizes[-1:]:
      self.dense_layers.add(tf.keras.layers.Dense(layer_size))

  def call(self, inputs):
    feature_embedding = self.embedding_model(inputs)
    return self.dense_layers(feature_embedding)
```

The **call** method returns the values calculated from feeding the input layer into the dense fully connected layers that have a *relu* non-linear activation function.

We also wrap the movie model in a candidate recommendation model:

```python
class CandidateModel(tf.keras.Model):
  """Model for encoding movies."""

  def __init__(self, layer_sizes):
    """Model for encoding movies.

    Args:
      layer_sizes:
        A list of integers where the i-th entry represents the number of units
        the i-th layer contains.
    """
    super().__init__()

    self.embedding_model = MovieModel()

    # Then construct the layers.
    self.dense_layers = tf.keras.Sequential()

    # Use the ReLU activation for all but the last layer.
    for layer_size in layer_sizes[:-1]:
      self.dense_layers.add(tf.keras.layers.Dense(layer_size, activation="relu"))

    # No activation for the last layer.
    for layer_size in layer_sizes[-1:]:
      self.dense_layers.add(tf.keras.layers.Dense(layer_size))

  def call(self, inputs):
    feature_embedding = self.embedding_model(inputs)
    return self.dense_layers(feature_embedding)
```

The **call** method returns the values calculated from feeding the input layer for a movie model into the dense fully connected layers that have a *relu* non-linear activation function.

We finally train a deep learning model by creating an instance of class **MovielensModel** and calling its inherited **fit** method:

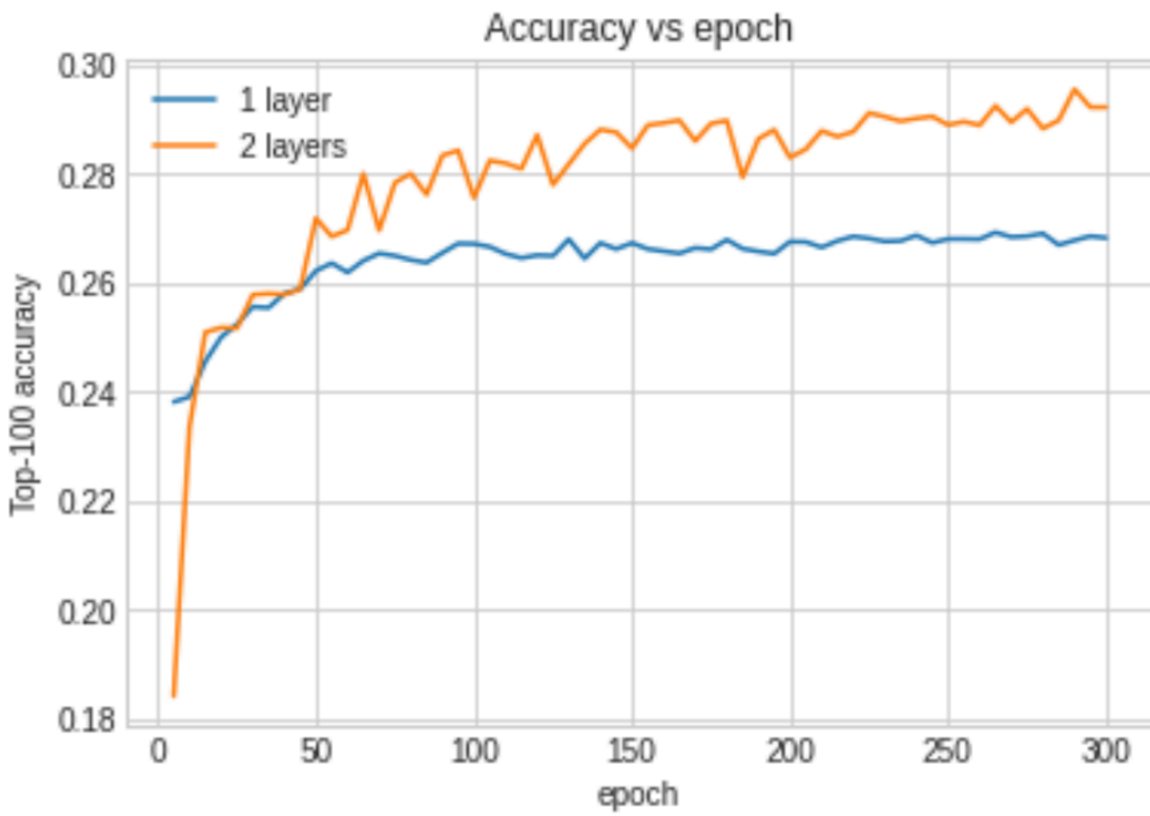
```
model = MovielensModel([64, 32])
model.compile(optimizer=tf.keras.optimizers.Adagrad(0.1))

two_layer_history = model.fit(
    cached_train,
    validation_data=cached_test,
    validation_freq=5,
    epochs=num_epochs,
    verbose=0)

accuracy = two_layer_history.history["val_factorized_top_k/top_100_categorical_accur\
acy"][-1]
print(f"Top-100 accuracy: {accuracy:.2f}.")
```

The output looks like:

```
Top-100 accuracy: 0.29.
```

This top-100 accuracy means that if you make a movie recommendation, it has a 29% chance of being in the top 100 recommendations for a user.

We can plot the training accuracy vs. training epoch for both one and two layers:



The example Google Colab project has an additional training run that gets better accuracy by stacking many additional hidden layers in the user and movie wrapper Python classes.

Recommendation Systems Wrap-up

If you need to write a recommendation system for your work then I hope this short overview chapter will get you started. Here are alternative approaches and a few resources:

- Consider using Amazon Personalize* which is a turn-key service on AWS.
- Consider using Google's turn-key Recommendations AI†.
- Eric Lundquist has written a Python library rankfm‡ for factorization machines for recommendation and ranking problems.
- If product data includes pictures then consider using this Keras example§ as a guide for creating embeddings for images and implementing image search.

*https://aws.amazon.com/personalize/
†https://cloud.google.com/recommendations
‡https://github.com/etlundquist/rankfm
§https://keras.io/examples/nlp/nl_image_search/

- A research idea: a Keras example* that was written by Khalid Salama that transforms input training data to a textual representation for input to a Transformer model. This example is based on the paper Behavior Sequence Transformer for E-commerce Recommendation in Alibaba† by Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou.

*https://keras.io/examples/structured_data/movielens_recommendations_transformers/
†https://arxiv.org/abs/1905.06874

Book Wrap-up

Thank you, dear reader, for spending the time for taking an adventure with me: exploring AI programming ideas using Python. Because some of the material will be dated quickly, I was motivated to write this book and release it quickly. Indeed, I wrote this book in just a two and a half month period.

My first ten books were published as conventional print books by McGraw-Hill, Springer-Verlag, John Riley, and other publishers. I then published several books with free licenses that can be downloaded from https://markwatson.com. My seven eBooks subsequently published on Leanpub* have cumulatively been updated many times to new editions. I would like to that the Leanpub platform for simplifying the process of writing and frequently updating eBooks.

I tend to remove some material in new eBook editions that might be outdated, and add new chapters on different topics. Because of this, when you get new editions of my books (free on Leanpub), consider also keeping the old editions.

I live in Sedona Arizona. If you are every passing through Sedona, contact me (see https://markwatson.com) if you would like to have coffee and talk about Python and AI.

*https://leanpub.com/u/markwatson